Virtualization with Limited Hardware Support

by

Bi Wu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Landon Cox, Supervisor

_____
Jeffrey Chase

_____
Alvin Lebeck

_____
Harvey Tuch

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2013

Abstract

Virtualization with Limited Hardware Support

by

Bi Wu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Landon Cox, Supervisor

_____
Jeffrey Chase

_____
Alvin Lebeck

_____
Harvey Tuch

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2013

Abstract

Over the past 15 years, virtualization has become a standard way to migrate old software to new hardware, isolate untrusted code, and encapsulate application state. However, many of the techniques for implementing common virtualization functionality, such as transparent isolation and full-system record and replay, rely on hardware features provided by the x86 architecture. As the mainstream computing landscape diversifies to include less feature-rich mobile and embedded systems, it is critical to rethink how to efficiently provide core virtualization functionality with limited hardware support.

As a result, this dissertation explores the feasibility of providing common virtualization functionality with limited hardware support.

We propose three approaches to virtualization using limited hardware support. First, we describe a way to transparently virtualize a CPU using hardware breakpoints and on-demand control-flow analysis. Next, we describe a way to record-and-replay virtual machine execution without relying on a hardware branch counter. And finally, we describe a virtual storage system that improves virtual machine I/O performance through a logging block store for inexpensive SD cards.

To CC, the person I loved so much.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Abbreviations

CFG	Control flow graph.
FTL	Flash translation layer.
GC	Garbage collection.
IP	Instruction pointer.
LBS	Logging block store.
MA	Machine address.
PA	Physical address.
PC	Program counter.
POI	Point of interest.
TLB	Translation lookaside buffer.
VA	Virtual address.
VM	Virtual machine.
VMM	Virtual machine monitor.

Acknowledgements

Words cannot express my thankfulness to my advisor, Dr. Landon Cox, for the constant support and guidance during my Ph.D.

And my sincerest appreciation to Dr. Harvey Tuch for mentoring and guiding my three internships at VMware.

1

Introduction

Virtualization is a technique that allows a piece of software to execute identically, barring timing effects, within a secured sandbox as it does on hardware while ensuring that the majority of the softwares instructions are directly executed on the CPU. In computer science, sandboxes are a security mechanism for separating running programs (Goldberg et al. (1996)). Web browser sandboxes, for example, can be used to run and render untrusted websites and prevent malicious code from compromising the operating system.

Virtualization provides code with the illusion of running in one execution environment, while in reality executing in another one. For example, kernel code may think that it is running in a privileged mode, when it isn't. In order to virtualize a physical machine, all the physical hardware on a typical machine must be virtualized. This includes CPU, memory, I/O, etc.. Virtualization is a powerful and well studied technique that can be used in a variety of ways. Over time, virtualization has evolved to provide security, encapsulation and isolation for multiple operating systems, allowing them to co-exist, run simultaneously and share resources on a single physical machine.

A virtual machine monitor (VMM), or a hypervisor, is a piece of software that provides virtualization functionality. The software that runs within the virtualization sandbox is the guest, or the virtual machine (VM). The software environment that the VMM runs within is the host. A Type 1 hypervisor runs directly on the physical hardware and has total control over its physical devices. A Type 2 hypervisor runs within a host operating system and does not have direct control over the physical devices, and instead emulates virtual devices with the assistance of the host operating system. Because the guests do not have direct control over physical hardware, they are presented with virtual devices emulated and managed by the hypervisor. A typical scenario for a virtualization environment is to have the guest operating system run on virtual CPUs and communicate with a virtual network adapter and virtual disk drives. The virtual network adapter can be bridged or directly connected to a physical network adapter and the virtual disk drives are usually stored as virtual disk images on the physical disks. A typical use case scenario involves VMMs running on physical machines within a data center, with the VMMs separating and managing many guest operating systems on the same physical machine, and managing the physical resources on it.

With virtualization emerged many useful and important functionality made possible by executing software under a VMM. One of them is record and replay, which is the ability to record the trace of a sequence of instruction execution and re-execute the sequence to produce the same output with the same input. This functionality is very useful, for example, when we can record the trace of execution of a virtual machine and produce the same result on another virtual machine and have a backup copy running in case the first one crashes. Other than providing fault tolerance, record and replay may also be useful in providing developers with better debug information by reproducing the sequence of events leading to the bug.

The most straightforward way of sandboxing an operating system is emulation. It

requires emulating or translating all the instructions of the sandboxed software into another set of instructions that can be executed within the sandbox. QEMU (Bellard (2005)) for example, emulates a machine-like environment for operating systems on many different architectures and enable them to run on the x86 architecture. It is much slower because the translation and emulation can be very costly when used on every single instructions of the target software, but it is among the very few ways to execute binaries intended for one set of CPUs on another set of CPUs with incompatible and different instruction sets. In Chapter 3 we will discuss the performance of QEMU and emulation in general.

In contrast, we can achieve much faster speed by having the majority of the virtual machine run natively without emulation. However, the virtualization sandbox usually means that the software running within the sandbox would run with a different, usually non-privileged, environment than a physical CPU, because in order for the VMM to provide the encapsulation and security properties, the sandboxed software cannot have direct control over the CPU. Because some privileged-mode-only instructions cannot be executed under non-privileged mode, the VMM must regain control just before emulation is required.

In the past, virtualization had been intensively studied on the x86 architecture and in the context of desktops and servers, and many virtualization implementations assume some hardware features specific to the x86 architecture. For example, a typical way of providing CPU virtualization, binary translation, requires architectural features such as segmentation, and in general, an execute-only mechanism (i.e. the ability for a region of memory to contain executable but not readable code) on the CPU. Record and replay implementations, for example, assume that the CPU has precise hardware branch counters. Beyond CPU virtualization, when moving beyond just desktop and server system architectures, devices exhibits different characteristics than desktops and servers, introducing complications when virtualizing. For

example, we are faced with disproportionately slow external storage devices that is used to store virtual machine images on mobile devices such as smartphones.

In recent years, as mobile devices started to become an essential part of everyday computing, virtualization on mobile devices has begun to emerge as a solution for supporting multiple profiles on the same device. However, many old ARM processors and embedded processors do not share the same hardware features as x86 processors. For example, even though very top tier devices are starting to be equipped with CPUs having hardware virtualization support, there are millions of existing and new devices without such support. ARM-based mobile and embedded systems do not have reliable hardware branch counters and usually do not have fast storage systems.

In this dissertation, we attempt to explore how to provide virtualization using a limited set of hardware features common on ARM-base mobile and embedded systems.

In Chapter 3, we attempt to provide a transparent CPU virtualization solution using hardware breakpoints and control flow analysis. Hardware breakpoints are usually CPU specific registers and the functionality to stop execution at a desired pointed set in advance by software. It is widely available on almost all architectures due to it being one of the few basic functionalities an architecture must provide to ensure that software running on it is debuggable. Control flow analysis is an analysis performed on software without actually executing it, which determines the order of which instructions are executed. We utilize control flow analysis to pinpoint instructions that cannot be executed directly within the virtual machine sandbox and trap them with the assistance of hardware breakpoints.

In Chapter 4, we explore the possibility of providing record and replay functionality on CPUs without hardware branch counters. We show that by inspecting the guest kernel internal states we can provide useful record and replay functionality

without strictly adhering to the sequence of recorded execution during replay.

In Chapter 5, we discuss a specific problem for virtualization on mobile devices. We will show that it is highly inefficient to store VMs virtual disk images using standard virtual machine image formats directly on a mobile devices external storage, which is usually an SD card. We will propose an alternative solution with much improved VM I/O performance.

As a result of this work, we will validate the following hypothesis: It is possible to efficiently virtualize and provide advanced virtualization functionality to a range of systems without relying on x86 and PC specific virtualization technologies.

2

Related work

2.1 CPU Virtualization

Virtualization, since its inception decades ago (Goldberg (1974)), has found its way into many areas of personal computing and cloud computing. The core concept of virtualization is to provide a sandbox for operating systems so that multiple operating systems can be run on the same machine without affecting each other. This conceptual sandbox provides important abstraction and isolation properties so that it is possible to abstract, share and manage resources on a single physical machine among many virtual machines running on the same hardware.

Ideally, we would like to have operating systems run within the sandbox without any modification (i.e. modification to the source code and recompilation). On architectures which are not classically virtualizable such as pre-VT x86 and pre-VE ARM, binary translation is the technique used to run unmodified guest operating systems with reasonable speed. Other types of virtualization, for example, paravirtualization and user/application level virtualization requires access to and modification of the source code of the guest operating system or applications.

One of the common problems facing virtual machine implementations is how to deal with the privileged mode instructions. Such instructions operate on privileged processors features such as page tables and I/O, and can only be executed within privileged mode without trapping or having incorrect semantics (e.g. as with sensitive instructions)(Goldberg (1974)). Xen (Barham et al. (2003)) talks about a widely used virtual machine and the idea of paravirtualization. Paravirtualization requires modification of the guest kernel source code so that these instructions are changed to hypercalls into the virtual machine monitor, whereas full virtualization does not require a modification of the guest kernel. Usually full virtualization is implemented using binary translation or with hardware virtualization support. Paravirtualization trades slightly better performance and a simpler virtual machine monitor for the inconvenience of modifying the guest kernel. UML (King et al. (2003)) takes a step even further to incorporate VMM modifications in both the host and the guest to eliminate most of the overhead of switching among host OS, the VMM and the guest to improve performance of the guest.

Hardware virtualization support is a CPU feature that allows a VMM to virtualize without having to deal with sensitive instructions. Without hardware virtualization support, the traditional way of implementing virtualization on x86 is binary translation. Bugnion et al. (2012) describes how binary translation is done on x86 using segmentation techniques. Adams and Agesen (2006) compares performance of binary translation against early hardware assisted virtualization on x86.

Cell (Andrus et al. (2011)) describes an Android level virtualization by emulating devices required by Android (such as timer, GPU, camera, radio etc.) using OS-level containers and namespace isolation and have virtual phones running a full Android user mode stack. This approach is very fast but is only applicable to Android.

Many other virtualization platforms use similar binary translation or binary rewrite techniques. Virtualization by emulation (Bochs (Lawton (1996)) for exam-

Table 2.1: Trade-offs among virtualization implementations

	Modification of guest required?	Speed	Hardware requirements	Implementation complexity
Emulation	No	Very slow	None	High
Hardware virtualization	No	Very fast	Hardware virtualization support	Low
Binary translation	No	Fast	Execute only mechanism	Very high
Paravirtualization	Yes	Fast	None	Low

ple) simulates the entire machine environment, decodes the machine instructions and emulates their result. It has the advantage of being able to execute cross-platform code, because the instructions are not directly executed but emulated. Another example is QEMU (Bellard (2005)), which provides a faster emulation by dynamically parsing and compiling instruction streams into native code.

While all virtual machine implementation techniques provide isolation and encapsulation properties, they trade among the need for kernel source modification, hardware requirements and performance (Table 1).

In addition, Turtles (Ben-Yehuda et al. (2010)) describes a nested virtualization architecture where virtual machines run inside a virtual machine. By emulating the Intel VT-x hardware virtualization extension, the outer level VMM can be aware of inner level VMM events and optimizes event processing and bypasses unnecessary levels of indirection such as page table translations.

With the prevalence of hardware virtualization support (e.g. VT-x on the x86 architecture), there emerges system research that attempts to use VT-x for purposes other than virtualization. For example, Belay et al. (2012) proposed using hardware virtualization support to expose kernel level primitives to applications so that they can manage privileged CPU features such as page tables and achieve faster performance.

2.2 Record and replay

Hypervisor-based fault tolerance (Bressoud and Schneider (1996)) describes the basic principles and ideas behind virtual machine record and replay. Record and replay for a virtual machine requires that the replayed stream of instructions match the recorded stream of instructions. In order to ensure such properties, one has to resort to hardware counters which count the number of instructions executed, to correctly intercept the guest execution before reaching the point of external event injection.

Record and replay had been mainly used as a tool to debug or provide fault tolerance. Revirt (Dunlap et al. (2002)) describes an implementation of record and replay on a UMLinux host with low execution and storage overhead for the record and replay process. It utilizes the hardware branch counters to facilitate the injection of external events during replay. The authors verify the correctness of the record and replay and establish its usefulness in analyzing security issues on the system.

Debugging operating systems with time-traveling virtual machines (King et al. (2005)) implements a record and replay framework integrated into GDB to log and debug the entire operating system and allows fast forwarding between checkpoints. Operating system debugging is a challenging problem due to the OS running directly on the hardware and it being very difficult to reproduce the sequence of events leading up to the crash. Record and replay with interleaved checkpoints solves these difficulties while allowing non-intrusive attachment to the OS from an external point of view.

Live migration of virtual machine based on full system trace and replay (Liu et al. (2009)) describes a VM migration system using checkpointing and recording the trace of the virtual machine during migration, which allows for very little downtime and reduced network bandwidth consumption.

Multi-processor record and replay share some similarities with the work we are

about to present, because the sharing of memory space requires that the implementation is able to handle the interleaving of memory access from different processors. Dunlap et al. (2008) discusses the problem of memory sharing in multi-processor record and replay and solves this issue by using hardware page protection to log the interleaved memory accesses. Respec (Lee et al. (2010)) implements an online record and replay that efficiently handles multiprocessor record and replay. It logs synchronization operations which should guarantee that most of the replay is correct. In the event that these operation logs fail to provide a correct replay, detected by deviations from the observed register and memory state, it uses a more stringent method of serializing the thread and logging the scheduling order.

Crosscut (Chow et al. (2010)) implements a selective replay system that can pick out or delete information from the replay log to preserve privacy. By using introspection into the VM on a process level or reflection by the system interpreter, it is able to tailor the replay into specific needs, such as removing a sensitive string from the log, or only replaying a specific process.

In Chapter 4, we focus on single processor record and replay, and the technical challenges of implementing on platforms without precise hardware performance counters.

2.3 Flash storage virtualization and log structured file systems

Bergmann and the Linaro Project (Linaro (2001); Bergmann (2011)) studied the performance of SD cards and tried to understand the intrinsics of the FTL and why it is slow to write non-seqential small blocks. They propose kernel level changes to improve the performance of file systems on SD cards.

Desnoyers (2013) is another nice summary of the intrinsic characteristics in design and implementation of flash storage devices. It describes basic parameters such as page and block size, read/write/erase speed etc. and how they interact with each

other and are limited by the size and cost of the flash device.

While other types of flash storage had been studied, for example, El Maghraoui et al. (2010); Saxena and Swift (2010); Rajimwale et al. (2009); Agrawal et al. (2008) on SSDs; Nath and Gibbons (2010) on Compact Flash (CF) cards; Bouganim et al. (2009); Birrell et al. (2007) on USB flash drives, SD cards had received very little attention perhaps due to the assumption that they are not used as key backing stores for important workloads such as virtual machine images. Without publicly available SD cards implementation details, it is unclear whether the result from these related work can be applied to SD cards.

Log structured file systems were proposed to address the I/O bottleneck caused by fast CPU and slow disks (Rosenblum and Ousterhout (1991)). A similar scenario exists when we use SD cards as the storage media on mobile devices. The Cloudburst project (Bartels and Mann (2001)) uses log structured file systems to speed up accesses on NOR flash chips.

Recently as smartphones start to become a force to reckon with in everyday computing, there has emerged research that study the performance of smartphone storage. Jeong et al. (2013) studies the performance of Android applications and achieve significant performance increase with the combination of journaling file system and smarter meta-data flushes. Another example is Kim et al. (2012) which identifies the randomness of Android applications′ I/O characteristics and proposed improvements such as improved hardware, RAID over SD and log based file systems.

3

Virtualization using breakpoint tracing

Building a virtual machine on legacy architectures with little to no hardware virtualization support presents significant challenges. Without hardware virtualization extensions, the traditional way of implementing virtualization is architecture specific. We present an alternative highly portable way of implementing virtualization using hardware breakpoints geared towards these legacy architectures, such as mobile and embedded systems, which feature non-x86 cores.

3.1 Introduction

Virtualization has witnessed a recent re-emergence on commodity x86 systems and evolved into many products and implementations. The first x86 hypervisors used binary translation to implement virtualization, and it had been the predominant full system virtualization technique until hardware support was introduced on the x86 architecture. As mobile platforms are getting faster and becoming more resource rich over the years, virtualization had found its uses there in supporting multiple mobile identities and possibly in the future, ARM clouds and servers. Up until now, ARM virtualization had been using paravirtualization for the most part, with Xen

on ARM (Hwang et al. (2008)) and the VMware Mobile Virtual Platform (Barr et al. (2010)) being prominent examples.

We first address the various challenges found in binary translations on x86 and ARM and then propose a uniform solution for all platforms that have very limited hardware support for virtualization.

3.2 Binary translation

In order to run a non-paravirtualized guest, it is essential that we need to be able to virtualize the guest binary without modifying the source code of the guest OS. When the guest is virtualized, it does not have direct control over the physical machine, and therefore runs at user mode privileges. However, there are instructions from the guest binary that would silently fail if they run at user mode privileges, meaning that those instructions do not produce the desired result and do not raise an exception. They are called *sensitive instructions* which need to be taken care of by the virtual machine monitor before the guest can be virtualized. The idea behind binary translation is to rewrite the guest binary into other instructions that produce the desired behavior of the sensitive instructions. However, another requirement for fully virtualized guest is that the host must be totally transparent to the guest. Certain parts of the system, such as memory or CPU registers, must be the same as if they are on a physical machine. Therefore the guest must not discover that certain unmapped virtual memory is actually mapped and reads as unknown data, or that a different binary is being executed when the guest reads the code it is currently executing (typically occurs during self checksumming or self modifying). Because binary translated code lives at a different virtual address space than the original code, it is essential that the memory where the translated code resides is unreadable to itself. In other words, we require an *execute-only* mechanism for binary translation to work successfully.

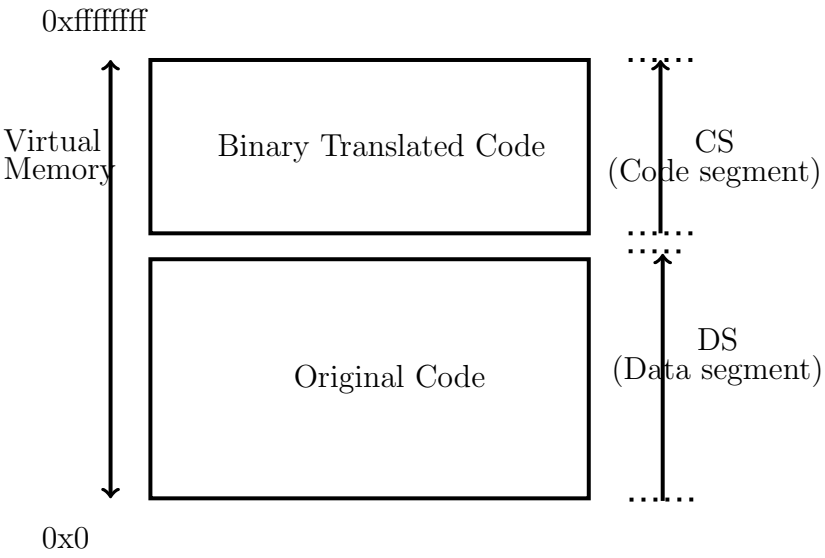

FIGURE 3.1: Binary translation on x86 using segmentation to implement execute-only mechanism

On the x86 architecture, we can achieve this requirement using segmentation techniques. Translated code runs at a different address than the original code. Since the translated code must not be able to read itself, it is place beyond the virtual address of the original code, and the segment CS (code segment) is set to the region of translated code′s virtual space so that the translated code can execute. But for all load/store memory accesses, which use DS (data segment) and other segments, we set those segments to only include the original code′s virtual address. In this way, if any code tries to read the translated code area, it will fault (Figure 3.1) (Bugnion et al. (2012)).

This solution is necessary because page protections are not sufficient to guarantee the property we seek. Both the translated code and original code are under user mode and belong to user pages, if they can be executed, they can also be read.

The same problem exists on ARM where we would like to hide the translated code from itself. Unfortunately ARM does not support segmentation. We have to accomplish this using the `LDRT` instruction. This instruction loads a memory

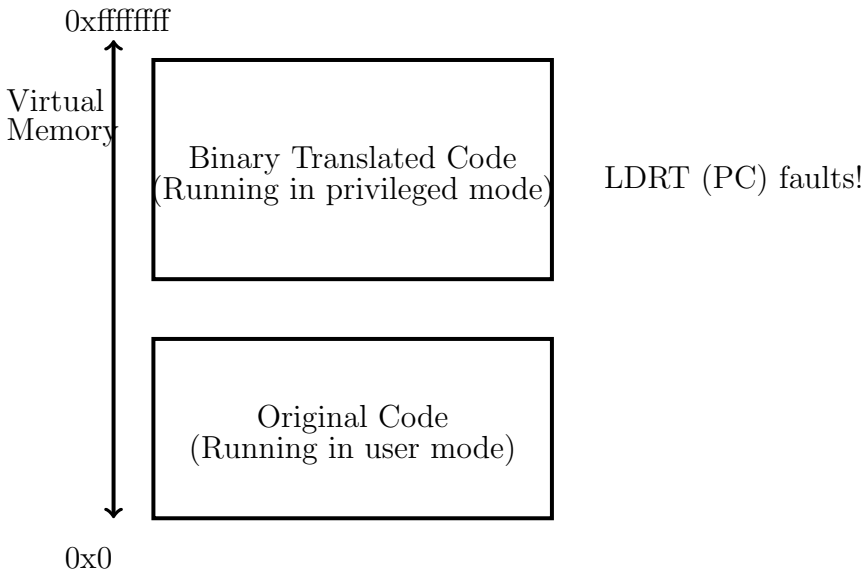

FIGURE 3.2: Binary translation on ARM using LDRT to implement execute-only mechanism

address into a register using user mode privilege. That is to say, if the memory address specified happens to be in a user page, it will fault. It is traditionally used by kernels to simulate a user mode access so that it can check whether certain memory addresses (usually from some parameters passed from user mode processes) are accessible by user mode or not. Using this instruction, we make translated code execute in privileged mode and all memory loads (`LDR` instructions) are translated into `LDRT` instructions. The pages are set so that the original code is in user mode and the translated code is in privileged mode. If any of the translated code tries to read itself using `LDR` instruction on the address of the program counter (PC), the corresponding translated `LDRT` instruction will fault (Figure 3.2).

Although this mechanism accomplishes the job, it has security implications because it puts a burden on the binary translator that the translated code must be bullet-proof in security. Even the slightest bug opens up the potential for a fatal attack from the guest which can then take control of the host system.

Other than the above two platforms mentioned, binary translation is difficult to

do under x86 64 bit because it does not support segmentation. If ARM did not have `LDRT` instructions, binary translation would also be hard to implement effciently on it. Overall, binary translation relies heavily on architecture-specific features. *It is not possible to apply a successful prototype from one type of CPU to another type of CPU.* For example, both MIPS architecture and PowerPC architecture lack a reliable way to virtualize without hardware virtualization support.

3.3 Breakpoint tracing

We propose that, by requiring hardware debugging breakpoints, we are able to implement virtualization in a new way without relying on an execute-only mechanisms which some architectures lack. Our implementation can be applied to any architecture with hardware debugging breakpoints.

In order to implement virtualization on CPUs with no hardware virtualization support, we need to be able to handle sensitive instructions by emulation. Because we would like a virtualized guest to run in an unprivileged mode with unmodified kernels, those instructions that normally operate under privileged mode must either yield the same result, or be emulated.

Our implementation traps sensitive instruction using hardware breakpoints. x86 and other CISC architectures have variable instruction lengths, and there are maybe hundreds of sensitive instructions in a kernel yet only 4-8 hardware breakpoints in any given CPU. Therefore it is necessary that we know which sensitive instructions are about to be executed.

In order to achieve this, we use dynamic analysis to construct the control flow graph (CFG) of the guest kernel. The CFG analysis identifies points of interests (POIs) which are defined as branches and sensitive instructions. We can then ask the question "given N number of breakpoints and a control flow graph with points of interests, where in the control flow graph must those breakpoints be placed?"

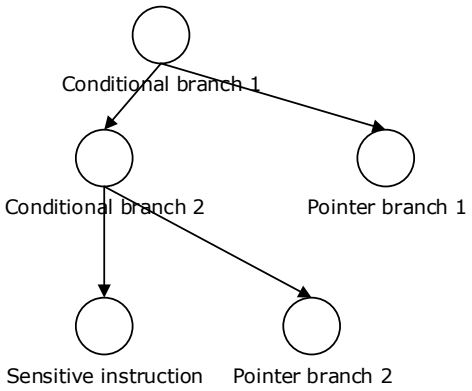

FIGURE 3.3: Guest control flow graph analysis

The objective here is to ensure that we place hardware breakpoints on appropriate places so that we trap all the sensitive instructions and maintain control over the kernel code flow. For example, if the analysis result is as shown in Figure 3.3, and suppose we have 3 available hardware breakpoints in the CPU, we have to place the breakpoints on the sensitive instruction, the `pointer branch 1` instruction and `pointer branch 2` instruction respectively. However if we only have 2 available hardware breakpoints in the CPU, we have to place the breakpoints on `conditional branch 2` instruction and `pointer branch 1` instruction respectively, and execute the guest until one of the breakpoints is triggered to determine what to do next. If we reach the `pointer branch 1` instruction, we need to perform further CFG analysis; if we reach the `conditional branch 2` instruction, we need to place the breakpoints on the `sensitive instruction` and `pointer branch 2` instruction. Also, if we ever reach a pointer branch instruction, we turn on single stepping before executing the next guest instruction and single step once to determine the position of the next executed instruction before the CFG analysis.

Our breakpoint tracing approach has several advantages over prior approaches. First, it can run an unmodified guest kernel. Second, hardware breakpoints are

a more common CPU feature than both hardware virtualization extensions and execute-only mechanisms. Third, this technique is simple and cross-platform. Fourth, our approach may have a security advantage over binary translation on ARM due to having no guest code running in privileged mode. The trade-off is a speed penalty due to more instructions being trapped than with binary translation or hardware virtualization.

3.4 Implementation

In this section we describe, step by step, how the breakpoint tracing virtual machine monitor is implemented and discuss some of the engineering challenges in the implementation. We implemented both the x86 and ARM version of this technique. We will focus on the x86 version during our elaboration because it is more complete and complex compared to the ARM version, and is also the version we will focus for our experiments.

The virtual machine monitor (VMM) implementation manages two *worlds*. The host world is the environment of the physical machine. The guest world is the environment of the guest emulated machine. The VMM is responsible for setting up and switching to the guest world.

The first problem we face is the need to choose the host OS and guest OS. A Linux host and guest is ideal for our implementation because it is open source and easy to debug. In theory any other host/guest combination is possible, and the architecture should support porting our implementation to another host or guest with ease. We choose C as the programming language for its portability. No matter what host we choose to run on, we need the processor's privileged mode access because at the very least, we need to be able to manage the CPU states such as page tables on the physical machine. Therefore, on a Linux host, we build the VMM into a loadable kernel module (LKM) to bootstrap the VMM's privileged mode access.

This bootstrapper code is host specific and if we are on Windows, the bootstrapper needs to be a kernel device driver.

The next problem we face is to design a guest world memory layout. The challenge here is that the guest world must have access to all virtual address space. A natural solution is to have a fresh, empty page table for the guest and have the VMM switch to the guest page table before beginning guest's execution. However, on almost all architectures the "switch page table" instruction must have a valid virtual page mapping on both the current executing page table and the target page table it's about to switch to. Therefore we have to "steal" some pages (we call these bridge pages) from the guest virtual address space so that the switcher code for the guest can be run within those virtual pages. However, the guest is free to access the virtual addresses of the bridge pages because it has no idea that it is running on a virtual machine. One possible solution to this conflict is to emulate instructions that refer bridge pages. However, we can dodge or emulate the rare conflicts by moving the switcher code to another virtual address if the guest requests access to those pages. As a consequence, we would like to choose some virtual address space high up in the 0xd0000000 - 0xffffffff area for our switcher code so that dodging rarely happens. Also, the switcher code must be positional independent which means it is able to run at any virtual address without modification (i.e. contains no relocation). Figure 3.4 describes the host/guest memory layout.

The general flow of execution of the virtual machine is as follows:

```
Bootstrap the guest
Repeat {
    VMM preprocesses the guest and set breakpoints, if necessary
    VMM switches to the guest world to begin execution
    Interrupts/faults stop guest execution
```

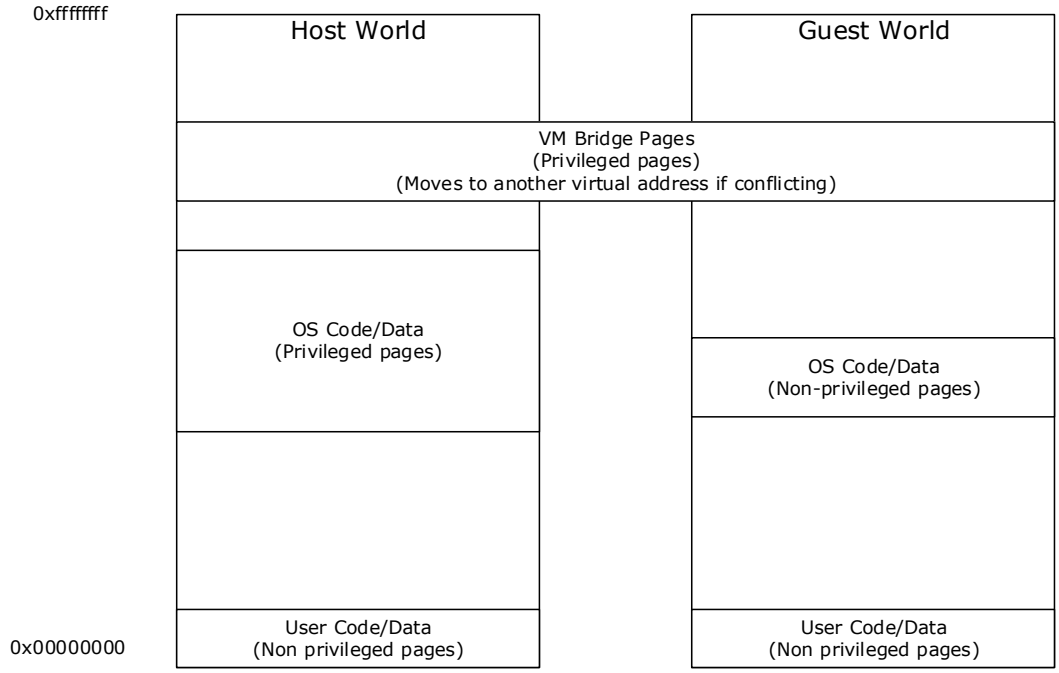

FIGURE 3.4: Virtual machine memory layout

```
    VMM handles interrupts/faults
}
```

On x86, we bootstrap the guest in the same way a physical machine would bootstrap an operating system when powered on. We load the boot sector of the boot disk (in our case, the floppy drive) at address 0:7c00, and set the guest's instruction pointer (IP) to 0:7c00 to begin execution. The guest boot sector would then perform the rest of the system initialization including loading the operating system and executing it just as it does on a physical machine. The VMM's job is to service all interrupts and faults generated by the guest after the bootstrap.

Because the guest has no direct control over the physical machine, any interrupt generated during guest execution is actually meant for the host. The VMM should simply re-deliver the interrupt to the host by re-asserting or re-triggering the interrupt while it's in the host world. When a fault occurs in the guest world, it can be

one of the following types: page fault, execution fault and breakpoint fault.

When a page fault occurs, it is either handled by the VMM's shadow page table handler (described later), or delivered to the guest. When an execution fault occurs, it means that either the guest executed certain instructions that cannot be executed under user mode (i.e. requires VMM emulation) or the guest has a true fault, in which case we need to deliver the fault to the guest. When a breakpoint fault occurs, the VMM needs to update its view of the control flow graph and determine the next set of breakpoints to place for our breakpoint tracing technique. Figure 3.5 shows the general flow of the VMM and how various components interact with each other.

In order to perform the functionality we have described, the VMM needs to keep track of all the guest states. The guest states are abstracted into a "world" data structure. Within the world structure, we maintain all guest register states and shadowed register states. We also keep a list of guest assigned pages and their corresponding mappings for VMM memory management. Emulated virtual device states are also kept for VMM device handlers. Figure 3.6 illustrates the data structure.

Next, we describe in detail each component of our VMM.

3.4.1 World switcher

To switch from the host world to the guest world, the host operating system calls into the VMM process to initiate execution of the guest. The world switcher context switches the processor from host world to the guest world and execute guest code directly without any modification. In addition to switching the page tables, we also need to switch the interrupt handling vectors so that when the guest takes an interrupt or a fault, it is the VMMs fault handler instead of the hosts fault handler that takes over. On the x86 architecture, we need to switch the global descriptor table (GDT), which is a data structure for describing segmentation, and the interrupt descriptor table (IDT) which is a data structure for describing exception handlers,

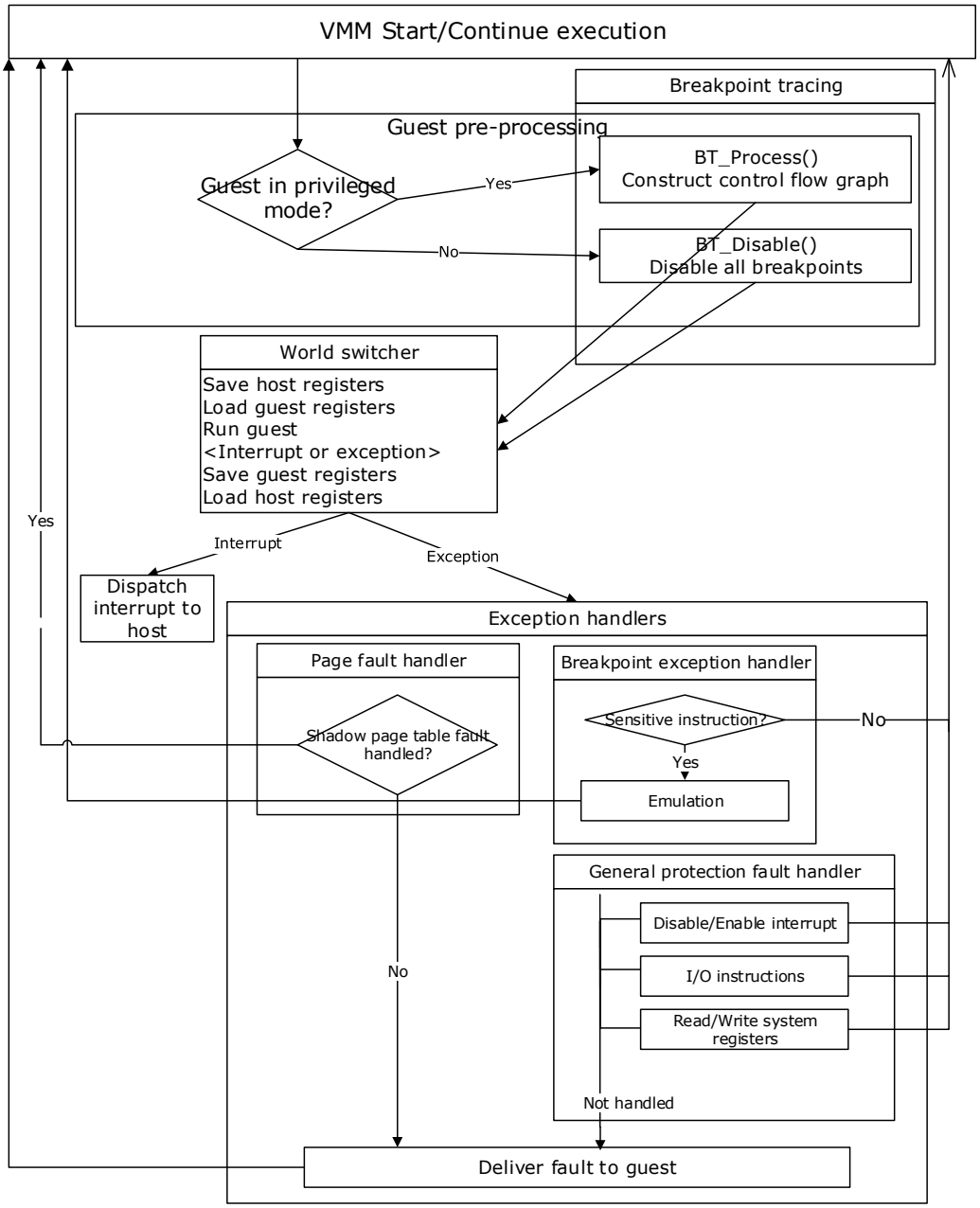

FIGURE 3.5: Guest world event flow

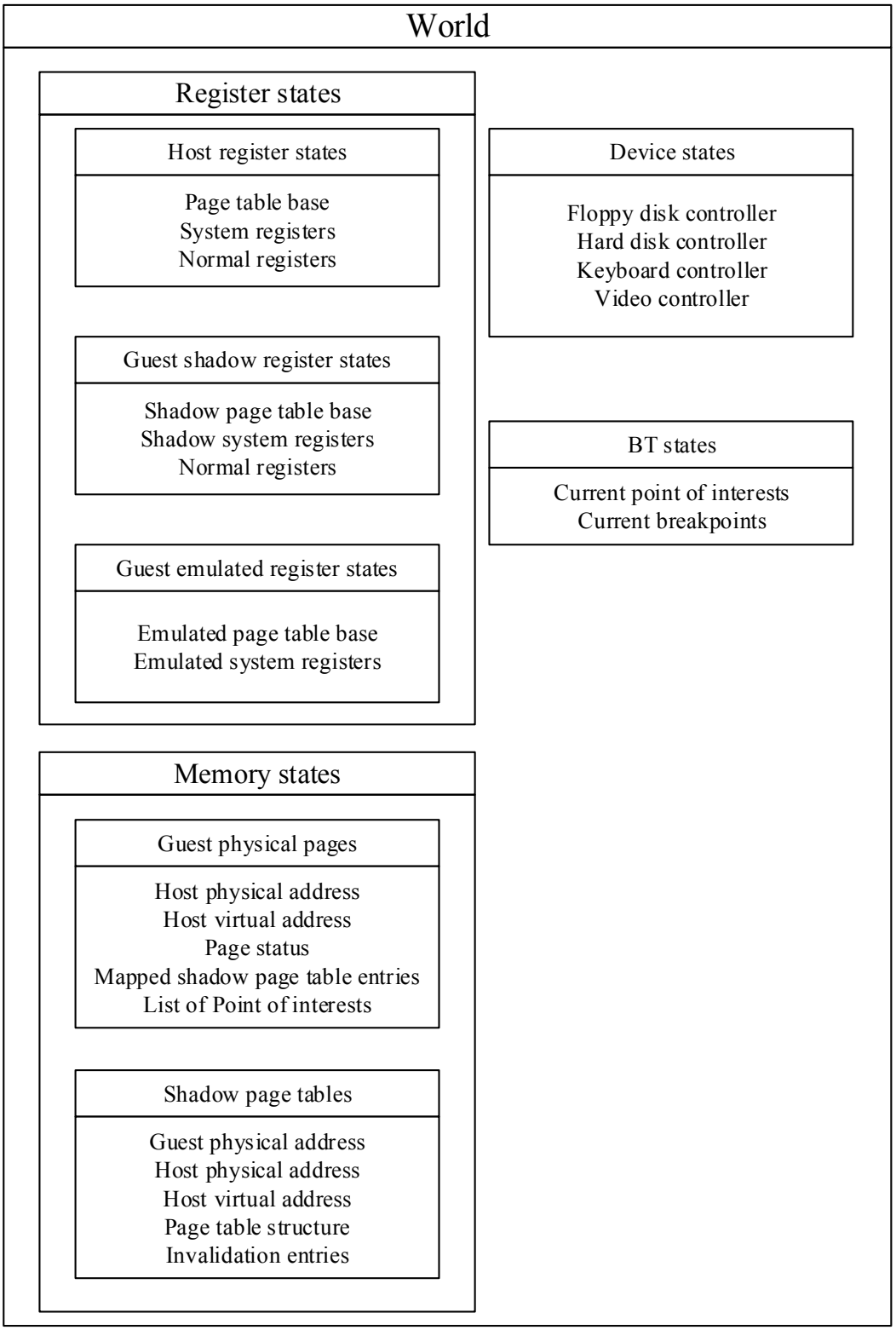

FIGURE 3.6: Guest world data structure

and the task register (TR), which provides information about current task and privileged mode stack position. On the x86 architecture, the world switch procedure is as follows:

```
Disable interrupts
Save all host normal registers
Save all host control registers (Page table base, GDT, IDT, TR)
Load guest shadow GDT
Load guest shadow IDT
In guest GDT, set guest TR not busy
Load guest TR
Load all guest normal registers
Load all guest control registers (Shadow page table base)
We are in guest context
Return to guest user mode using IRET
```

When an interrupt or fault is taken during guests execution, it traps to the handlers specified by the VMM instead of the hosts. A CPU fault is not host bound and requires VMM handling, with an interrupt it is required that we re-deliver the vector to the host. Upon taking a fault or interrupt, the fault handler does the following steps:

```
Save all guest normal registers
Restore all host normal registers
Restore host page table
Restore host GDT
Restore host IDT
In host GDT, set host TR not busy
Load host TR
```

```
We are now in host context, resuming kernel module execution
```

After the world switch back to the host, we are in host world and can access all host system functions and memory.

3.4.2 Guest memory management

Guest memory is managed by shadow page tables. Whenever a guest attempts to load a page table base register, it triggers a fault, enabling the VMM to intercept and perform appropriate actions. The guest is not aware of the host address space and believes that it is running on a physical machine. Therefore, it maps its virtual address (VA) to physical address (PA). It is the job of the VMM to translate guest physical addresses into host machine physical addresses (MA) (Figure 3.7). Therefore, whenever the guest attempts to load a page table, the VMM intercepts and supplies the CPU with a shadow page table instead. The shadow page table initially has no guest specified mappings. Therefore, virtual address accesses fault into the VMM. The VMM then checks the guests page table to determine if the fault is caused by the VM not mapping the pages correctly in the shadow page table (i.e. a *hidden fault*), or if the fault is caused by the guest not setting up a mapping correctly (i.e. a guest fault). Hidden faults are handled by the VMM without guests knowledge. Guest faults are delivered to the guest vector set by the guest IDT.

The guest OS frequently changes page tables and issues TLB-flush instructions to make the changes visible to the hardware. The VM needs to update the shadow page table accordingly as well when the guest flushes the TLB. TLB flush instructions sometimes flush specific virtual address within the guest, but most of the time, the guest issues an instruction to flush the entire TLB during process switch. A naive implementation would need to clear the entire corresponding shadow page table because it would have no idea of which entries of the guest page tables are updated. This naive implementation has huge performance issues because a new and empty

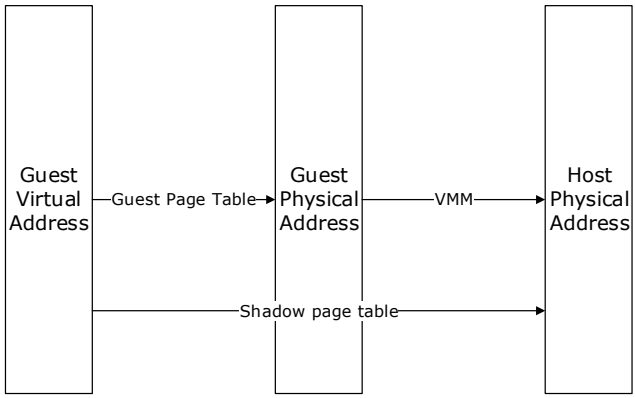

FIGURE 3.7: Shadow page table

shadow page table triggers many hidden faults. In order to improve performance, we write-protect the guest page table pages whenever the guest flushes the TLB. During guest execution, if the guest page table pages are written, the VMM removes the write protection and mark the corresponding shadow page table entries changed. The next time the guest issues a TLB flush instruction, the VMM would only need to clear the pages that had been marked as changed.

One additional optimization for our shadow page table implementation is optimizing the root page table. Purging a root page table in the corresponding shadow page table because the guest version was changed (write protection broken) is a very costly action. Since we do not know which entries were modified, we have to purge all links to the second level page table in our shadow page table root page which causes massive amount of hidden page faults later on. We optimize the root page table by copying the entire root page (4K in size) and compare the entries one by one if the write protection is broken. We do not purge the entries that were unchanged and therefore save some hidden faults from occuring.

3.4.3 Breakpoint tracing

In order to perform dynamic analysis of the guest kernel, we build a generic breakpoint tracing framework. The framework consists of three parts: decoder, control flow graph generator and breakpoint tracing plan builder.

The decoder decodes guest kernel code starting from the current program counter. It is necessary to perform dynamic analysis because the x86 architecture has a variable instruction length, and it is difficult to determine instruction boundaries until the code is about to be executed. The decoder marks certain instructions into POIs which are tagged into the following categories: conditional branches, pointer branches, unconditional branches, sensitive instructions, and unrecognized instructions. In order to avoid having to repeatedly decode the same encountered PC, we cache the decoded result and write protect the code page. We can find out what type of instruction the current PC has, by indexing into the cache using the PC. We do not have to perform decoding again as long as the write protection is not broken. The control flow graph (CFG) generator generates directed graphs that represent relationships among POIs. Conditional branches have two connected nodes that represent taken and not-taken branches. Unconditional branches have one connected node that represents the branch target. Pointer branches, sensitive instructions and unrecognized instructions have no connected nodes because the next instruction that executes afterwards can be an arbitrary instruction.

The breakpoint tracing plan builder builds a breakpoint plan from the given POI and determines on which POIs the breakpoints should be placed so that the VMM does not lose track of the guests execution, and that all sensitive instructions are trapped when executed. The plan builder uses a simple breadth first search algorithm to determine where to place the breakpoints given a CFG:

```
Total_Available_Breakpoints = 4
```

```
foreach Node in CFG
  Node.visited = false
BT_Breakpoints(Current_Executing_Node)



function BT_Breakpoints(Node)
  if (Node.visited) return
  Node.visited = true
  if (Total_Available_Breakpoints == 0) return

  switch(Node.Type)
    case UNCONDITIONAL_BRANCH:
      BT_Breakpoints(Node.Next)
      break

    case POINTER_BRANCH:
    case SENSITIVE_INSTRUCTION:
      Place_Breakpoints(Node)
      Total_Available_Breakpoints--
      break

    case CONDITIONAL_BRANCH:
      if (Total_Available_breakpoints == 1)
        Place_Breakpoints(Node)
        Total_Available_Breakpoints--
      else
        BT_Breakpoints(Node.Left)
```

```
        BT_Breakpoints(Node.Right)
      break

  if (Total_Available_Breakpoints == 0) return
end_of_function
```

When we receive a breakpoint fault during guest execution, we can determine where the guest has executed to in the CFG and perform the appropriate action. If we arrive at a pointer branch, we need to single step once to determine the next instruction to execute before performing CFG analysis. If we arrive at a sensitive instruction, we need to emulate the instruction and then continue the CFG analysis. If we arrive at a conditional branch, we simply perform more CFG analysis.

3.4.4 Guest side monitor

The world switch procedure we described is very expensive because in addition to an interrupt entry/exit, it flushes all TLB entries and reloads all other system registers (such as GDT and IDT on x86). The extra overhead from the world switch can cost a huge amount of CPU cycles equaling several thousands of instructions. In order to minimize the cost of world switching, we move some of the VMM code to the guest side. By stealing a virtual address not used by the guest, we map some of the exception processing code into the guest space as a monitor, and modify our exception handler to jump to the monitor code to be processed first. We avoid the cost of world switch and only take the cost of an interrupt entry/exit if the monitor is able to handle the exception. Currently in our implementation, we are able to handle most of the breakpoint exceptions and most of the privileged instruction faults. We only save a minimum amount of registers required for the monitor code to work, i.e. all the normal x86 registers and DS/ES segment registers. Note that it is possible for

the monitor to page fault, for example, if the monitor is trying to read the current guest stack and it happens to be unmapped. When the monitor faults, we have to fall back to the host world handler to deal with the event because we cannot call host system functions such as `mmap` from the monitor side, which is in the guest world. Monitor faults are detected by misconfigured stacks on exception entry. When an exception is triggered from the guest code (user mode privilege level), the stack is set to a fixed location defined by our VMM. However, when monitor code is running, the stack is not at that location, and when it faults, the stack remains the same because x86 doesnt change stack pointer if no privilege level switch occurs (the monitor is running at system mode privilege level). When the fault is detected, we simply reset the stack to the fixed location and proceed as if the monitor code was never run. With the monitor code installed, the general flow of the exception handler looks like:

```
x86 IDT Entry:
//the first item on the stack is the interrupt number
push $int_number
//save all registers
pusha
//DS and ES are required for C code to work
push ds
push es
//are we handling interrupt 1 (breakpoint exception)?
cmp $0x1, 40(%esp)
jne 1f
monitor_bt()
1:
//is the stack correctly aligned?
```

```
cmp $stack_location, %esp
jne 2f
//nope, the monitor faulted. Reset stack
mov $stack_location, %esp
2:
push fs
push gs
//save additional segment registers
//continue with the rest of the world switch
...
```

Note that because it is possible to fault within the monitor, the monitor code must be written with such a restriction in mind: The monitor must not modify a permanent status (i.e. a status variable stored in the `world` structure) of the guest and then faults. For example, the following monitor code has a bug:

```
if (world->poi->type == CONDITIONAL_BRANCH) {
    world->poi = world->poi->next;
    unsigned char *instruction = *(world->guest_regs->ip);
```

The above code would bug because we modify a permanent status of the guest world (POI) before fetching the currently executing instruction. However, such fetching action can page fault and therefore we can be left with a partially modified world state.

3.4.5 I/O

In order for the guest to interact with the virtualized environment, the VMM must implement basic I/O devices for the guest to use. We implement text console, key-

board, floppy drive and hard drive on x86, and for our ARM implementation, only the serial console is implemented. The guest OS (Linux, for example) can interact with these virtualized devices using the same unmodified drivers as on any other physical machine. On a physical machine, an operating system outputs to the screen by writing to the video cards memory buffer. On x86 if we are only concerned with text mode (i.e. 80x25 characters per screen) the buffer is located at physical address 0xb8000. We build a `ncurse` window that directly map guests screen buffer pages to our `ncurse` window so that any time a guest writes to its screen buffer, the result is reflected in our `ncurse` window. This window is the primary channel to observe and verify that the guest output.

In order for the guest to receive user input, we implement and expose an `AT keyboard` device to the guest OS. The keyboard interface on x86 is I/O ports 0x60 and 0x64. Whenever a key is pressed, an interrupt is generated on a vector specified by the guest. When the guest receives the interrupt delivery, it checks the keyboard I/O ports to read in any pending key presses as scancodes. The scancodes are then translated into actual keys. The VMM needs to translate the key inputs received from the `ncurse` window into scancodes and emulate passing the scancodes to the guest.

We also implement a floppy drive controller and a hard disk controller as the storage devices for the guest. The guest submits requests to read the disk via I/O commands, identifying which cylinder, head, and sector (CHS) it wants to read, and on which physical address it wants to save the results. The guest then may run other code while it waits for an interrupt signaling the completion of the I/O. The VMM, upon receiving the I/O commands, performs the I/O by reading the virtual disk image and modify the guest physical page with the I/O results before injecting the interrupt into the guest.

3.5 Experiments and discussions

In order to evaluate our breakpoint tracing technique, we engineer a proof of concept virtual machine monitor on both the x86 architecture and the ARM architecture. The ARM version is verified to be able to boot an unmodified Linux 3.4 guest on a Pandaboard ES. However, since there are more VMM implementations available on x86 for comparison, and also because it is easier on x86 to debug and analyze the performance of our implementation, the rest of the section will focus on experiments and analysis done on the x86 architecture. The test environment we use for the host is a Linux 3.4 kernel (SMP disabled) running on Intel Core 2 Duo processor with 4G memory. The guest is an unmodified Linux 3.4 kernel with necessary floppy disk and hard disk drivers. We compare our performance against QEMU and VirtualBox running on the same host using the same guest. We disable Intel VT-x on the BIOS and the QEMU/VirtualBox configurations to make sure we are making a fair comparison.

We evaluate our work to answer the following questions:

1. What is the overhead of the implementation?

2. Where does the overhead come from?

3. Can the implementation be faster?

In order to evaluate our implementation on CPU virtualization performance, we first run several micro-benchmarks containing critical kernel mode only workloads. Then, we run some application benchmarks to determine how our implementation would perform running real applications. Table 3.1 lists the experiments we run.

The micro-benchmarks demonstrate the worst case CPU virtualization overhead for our implementation. The majority of the overhead comes from two categories, the monitor itself for handling the breakpoint tracing and mode transitions (i.e. world

Table 3.1: Evaluated experiments

Experiment name	Experiment detail
getuid	`getuid()` syscall 10000 times.
forkexec	`fork()` and `exec()` a null program 10000 times.
process switch	Create two processes and ping-pong between them 1000000 times.
thread switch	Create two threads and ping-pong between them 100000 times.
segv	Triggers `SIGSEGV` 1500000 times.
gzip	`gzip` a file 1M in size with random data generated from `urandom`.
gcc bzip2.c	`gcc` the source file of bzip2. cold run is the first run while hot run is the second run without rebooting.
dd 10M	Run `dd` with source as `/dev/zero` and destination as the hard drive, copying 1M block size 10 times.
python computation	A Python benchmark that solves NQueens.

switches and monitor exits/entries). World switches are very costly as demonstrated by the results without a guest side monitor. The `getuid` experiment is special among all the experiments because it barely has any kernel mode code to execute before returning the UID result to the user mode code. Therefore it measures entirely the cost of guest mode switch. There are hardly any monitor overhead in this experiment and all the overhead comes from world switch. The reason that we are unable to handle everything within the monitor is because guest mode switching and page faults require read/write of the guest page table and shadow page table, which was not yet implemented. However, in theory, after we are able to fully handle all faults inside the guest-side monitor, we could reach about 30x overhead on the experiments other than `getuid` and near native performance on `getuid`, based on only counting the overhead of guest side monitor and the exception cost in our micro-benchmarks experiment.

The application benchmark shows that we are faster than QEMU in most cases. This is because QEMU is running as an emulator without KVM acceleration and is translating and compiling every single instruction. For application benchmarks, our

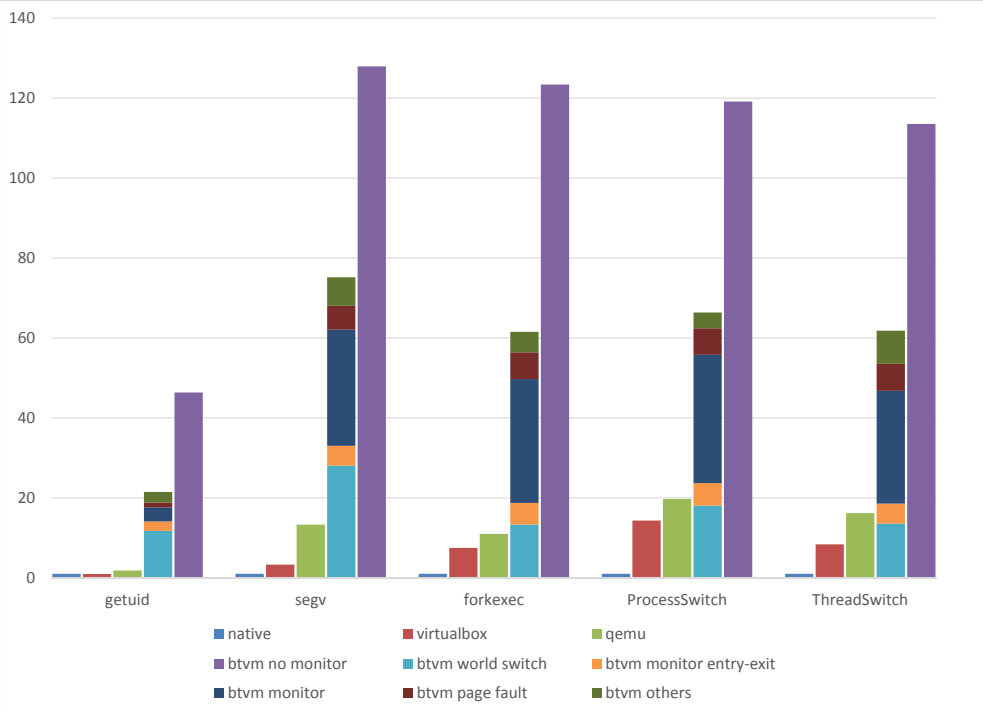

FIGURE 3.8: Micro-benchmarks result

implementation is closer to VirtualBox when less I/O is involved in the experiment while having considerable overheads when I/O is involved due to two factors. First, I/O instructions executed by the kernel traps and are not handled by the guest side monitor in our current implementation and therefore has to involve a world switch. Second, I/O from the guest requires execution of some guest kernel code at which our implementation is slower by comparison. For CPU intensive tasks, for example the hot run of gcc (much less I/O than the cold run) and python computation benchmark, our implementation is much closer closer to native and binary translation.

Lastly, because currently the ARM version of our implementation does not have a guest side monitor, we examine how the implementation could perform if fully ported to ARM platform by identifying and measuring some key costs of CPU virtualization.

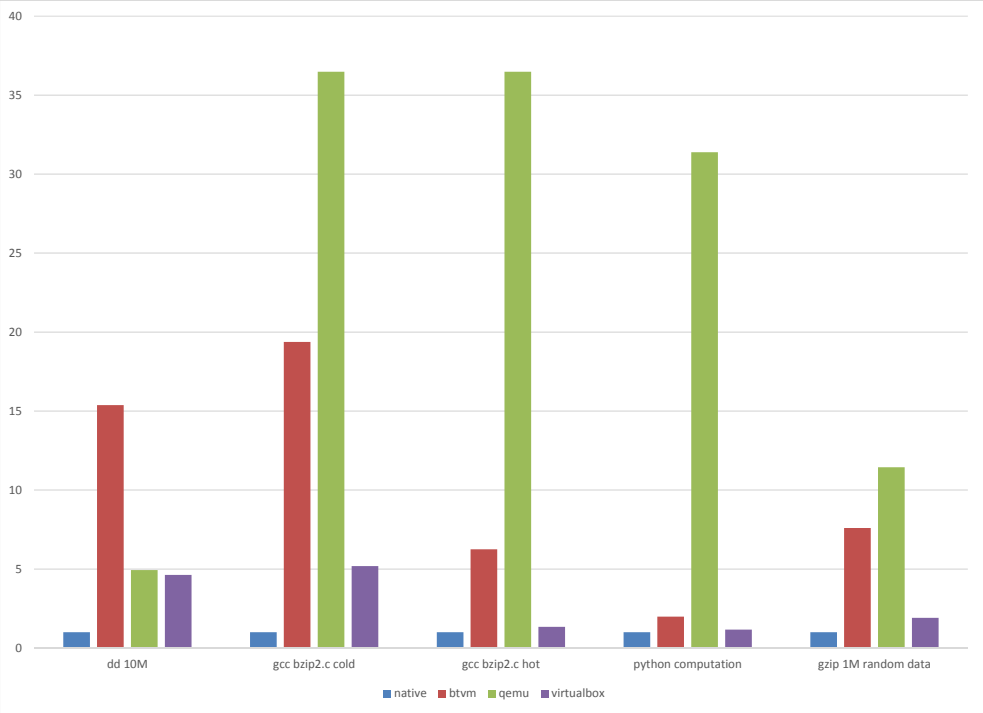

FIGURE 3.9: Application benchmarks result

The major overhead our implementation encounters is the world switch cost and exception cost. We measure what the costs of these operations are on both x86 and ARM. The world switch costs are measured with timed 10000 world switches in our implementation. The exception costs are measured with timed 10000 breakpoint exceptions. Both experiments are set up by running a null guest and placing a breakpoint on the first instruction that is about to be executed, therefore causing the guest to immediately trap and bounce back to the host or the monitor, where we time the start and end of the experiment by reading the processor time stamp counter. We compare it against a baseline of a totally empty function, and a baseline of a function that saves all normal processor registers and load all normal processor registers inside the function. The baseline of a function with register saves and register loads is a

FIGURE 3.10: World switch and exception costs

more fair comparison because it is practically impossible to engineer the guest side monitor or the host fault handler without register saves and loads. These experiments demonstrate that,

1. Relatively speaking, compared to function calls, it is much slower to handle an exception on x86 than on ARM. We have no exact proof as to why this is the case, but an educated guess suggests that ARM exceptions only trigger two banked register swap and a mode change before executing the fault handler, and all of these state changes involve only the CPU, whereas on x86, the CPU needs to read the IDT and TR to determine the exception handler's location, with all the data structures involved in this transition residing in memory. Also, another possible cause is the difference on CPU microarchitectures such as shorter pipeline length.

2. Relatively speaking, compared to exception handling, it is much slower on ARM to do a world switch, due to the fact that world switch on ARM does a lot more compared to its relatively simple exception handling procedure. It has to do a TLB flush, exception vector reset and mode stack save/load. As a consequence, it is utmost essential for the implementation to have a guest side monitor on ARM.

In all, our implementation can be faster by implementing a more thorough guest side monitor handling I/O and many other boundary cases such as guest privilege mode exit/entry. Also, an ARM version would have a slightly better performance relatively speaking, because the guest side monitor are less expensive to trap to compared to the x86 architecture.

3.6 Conclusion

In this chapter, we examined many virtualization techniques, especially binary translation, on x86 and ARM architectures, and discussed why it is difficult to apply existing binary translation techniques to mobile devices. We proposed a uniform solution for both architectures using hardware breakpoints and built a virtual machine using this technique. We examined various engineering challenges in building such a prototype. We performed several experiments to discuss the feasibility of this approach compared to existing binary translation techniques. We broke down the overhead of our approach and analyzed the fundamental architectural cost of world switches and exceptions.

4

Record and Replay without hardware counter support

Record and replay provides important functionality for debugging virtual machines and fault tolerance. However, this function cannot be easily implemented on the ARM architecture due to a lack of precise hardware counters. We propose an alternative implementation for paravirtualized guests and introduce a corresponding alternative concept of correctness for the replay. We discuss how the correctness of the replay can be guaranteed in general and in the special cases of signals and shared memory, and show that the overhead of this approach is reasonable.

4.1 Introduction

System virtualization dates back to the 70s where it first appeared on IBM 360/370 hardware (Goldberg (1974)). It took off rapidly in the last two decades with efficient and useful products on the x86 architecture, and found its way into data centers as an indispensable tool for cost effective computing. Contemporary virtualization research and industry efforts have had an x86 focus. In recent years, as CPU hardware

continued to improve, virtualization has been extended to platforms other than x86, and attempts have been made on the ARM architecture to implement virtualization, most notably VMware Horizon Mobile, which is a paravirtualization solution for ARMv7. The vastly improved processing power and capabilities of ARM CPUs motivates virtualization, since they are now capable of handling large memory sizes and numbers of cores, comparable to those in the desktop environment.

Record and replay is an integral part of the virtual machine functionality on a virtualization platform. It enables debugging and fault tolerance of virtual machine guests. Naturally, the question of how to implement record and replay on the ARM architecture arises, and it is more challenging than on x86 due to limited hardware support.

Record and replay takes a virtual machine image and recorded actions as input, and generates an output of virtual machine state as if those actions are done on the virtual machine in the time and sequence they were recorded. The output state is deterministic, a feature which is essential to debugging and fault tolerance of the guest VM. On x86, record and replay is implemented using an interpolation of program counter (PC) and processor hardware counters to pinpoint an exact point of injection of external events. The primary issue on ARM is the lack of precise hardware counters that supports this style of record and replay implementation.

We propose an alternative implementation on a paravirtualized mobile virtualization platform using only minor modifications to the guest kernel and virtual machine monitor (VMM). We also propose a different type of replay correctness than that used in traditional x86 implementation. We relax the traditional definition of "instruction by instruction" correctness and show that we can still achieve deterministic and observationally equivalent results in the end.

4.2 X86 record and replay implementation

Ultimately, in order for a replay to be correct (and useful), the sequence of observed guest events, namely, those generated by the replay must match those generated by the recording. In addition, the same guest state must be arrived at by the end of the replay. In other words, any record and replay implementation must guarantee that the result of the replay is deterministic. Given a fixed sequence and timing of external events, a replay of those events must arrive at the same result.

In order to guarantee such correct replay, we need to record all non-deterministic events (i.e. interrupt and interrupt state) as well as all input results, which are essential to reconstructing the events and I/O during replay. With this information recorded, a record and replay implementation must then guarantee replay correctness by carefully injecting the recorded events.

On x86, we guarantee replay correctness by having the replay execute the same sequence of instructions as was executed in the recording, and injecting external events at the same instruction as was recorded. In other words, we need to find the same instruction where external events are injected by carefully advancing the execution of the guest virtual machine until we arrive at such point.

Normally, there are only two ways to advance and capture the execution of instructions. Either a breakpoint is placed on the desired instruction, or single stepping is engaged and instructions execute one at a time. Both of these methods are slow to use. The first method of placing a breakpoint might seem fine in many situations, but suppose we have a loop in the execution path and the breakpoint is within the loop, it would take as many breakpoints traps as iterations to find the actual injection point, e.g.

```
for i = 1 to 10000 do
    check_for_input(i);
```

```
end
```

Fortunately, we can utilize hardware branch counters on x86 to solve this issue. x86 hardware branch counters provides an accurate count of how many branches have been executed since the last specified point of execution, and an interrupt can be set to trigger upon reaching a predefined number of branches executed by the CPU.

With the assistance of hardware branch counters, we can advance instructions during replay without resorting to the two slow methods aforementioned. The number of branches executed between each two consecutive events are recorded, and we only have to set the interrupt to trigger upon reaching the recorded number of branches, at which point we can simply place a breakpoint on the desired instruction and not have to worry about reaching the breakpoint multiple times, because the first time such breakpoint triggers is the point of injection we require.

4.3 Record and replay on ARM

Record and replay on ARM cannot be implemented in the same way as it is done on x86. ARM platforms do not have architecturally guaranteed accurate processor counters for use (ARM (2001)). We argue that we can relax certain aspects of the record and replay implementation on ARM and still produce a useful implementation with reasonable speed.

In order for a record and replay implementation to be useful, it must guarantee that the result of the replay is deterministic. I.e., given a fixed sequence and timing of external events, a replay of those events must arrive at the same result, which is measurable by the output and state of the virtual machine. We call such a record and replay implementation *logically correct*.

x86 record and replay implementations inject external events at the same instruc-

tion as was occurred in the recording, therefore producing *literally* the same stream of instructions executed during replay, ensuring correctness. We can however, relax this requirement, and not require a literal reproduction of the same instruction stream, while still producing a *logically* correct output, thereby satisfying the usefulness of the record and replay implementation.

The fundamental insight we present here comes from an important observation. Usually, the challenges for record and replay on any platform is how to manage incoming non-deterministic events potentially occuring at any guest instruction. Previous research has focused on how to pinpoint the particular instruction where these events occur. However, if we examine how some operating systems work, such as Linux, and how user processes interacts with the kernel, we can make the key observation that when such events occur, *usually* they do not affect state and execution of user processes. In other words, most of the time, when those events occur, they neither are observable by user processes nor impact user processes' state. For example, if an interrupt for a keypress occurs in user mode, the kernel processes the interrupt and records the keypress in some internal kernel buffer, and then may return to user mode. However, the event handling may make no change to any of the user processes, assuming no user processes are waiting asynchronously for a keypress. An important observation is that such events only affect kernel state, and during replay, the point of injection of such events often does not affect the state of the user process as long as the sequence of events is respected. In other words, such events can be injected anywhere in user mode as long as the injection occurs after the previous recorded event and before the next recorded event. Generally speaking, the vast majority of the interactions between kernel and user processes are through system calls. We have comprehensively examined the POSIX API, and other than using signals and shared memory, which we will discuss later, there are no other methods by which a process may interact with another process. In other words, if we exclude signals and

shared memory, between two system calls, if an interrupt occurs, there is no POSIX API whose result is dependent on the exact position of the instruction where the interrupt occurs. And if we exclude signals and the case of `longjmp` call, the program flow is deterministic regardless of interrupt occurrence, with the caller always returning to the next instruction to execute. Such an important observation enables us to implement a record and replay framework that can disregard the exact point of interrupt occurrence in most situations. Note that the random number generator of guest is deterministic between record and replay because the number generated is based on the sequences of external interrupts and time, which are identical between record and replay.

In order to maintain logical correctness of the replay, the VMM is made aware of the distinction between guest kernel and guest user processes. In many cases, even though a non-deterministic event occurs, it only affects the kernel and not any of the user processes. In order to guarantee that non-deterministic events are injected at the same point if such events happen in kernel mode, we modify the VMM to prohibit non-deterministic event injection unless the kernel calls "wait for interrupt"(WFI). This modification however, assumes the kernel will be regularly calling WFI or entering user mode, which, for example, is the case in Linux. This is a reasonable assumption because a paravirtualized guest provides the opportunity to ensure this regular invocation of WFI. We also record all the kernel's interactions with the VMM and devices, such as any timer or wall clock calls. In this way, we ensure that any timer or device related handling by the kernel, such as locks and scheduling decisions, is always the same in replay because the injection point of all external events are identical between record and replay, at the expense of increased interrupt latency depending on the type of applications and the system.

Although we have argued that for non-deterministic events, in many cases, a precise point of injection is either guaranteed (in kernel mode) or not required (in

user mode), there are cases where an exact point of injection is required. For example, shared memory and signals require a precise point of injection because, in these cases, non-deterministic events do affect the state of the user process. For example, if a `SIGALARM` occurs due to a timer interrupt, both the kernel and the user process are affected. The kernel takes the interrupt and is affected in terms of stack, handlers and randomization pool; the user mode process experiences a PC change (i.e. program flow change). Another example is if two user processes communicate via shared memory and a task switch (due to a timer interrupt) occurred, a precise injection is required because the processes involved must be replayed with the same view of the shared memory as was recorded.

Figure 4.1 illustrate the cases with signals.

There are three possible scenarios with signals in the guest.

1. An interrupt occurs during a process execution. Guest switches to kernel state. Signal occurs on the process. The process continues to execute with a different PC (signal handler).

2. A system call or fault occurs during a process execution. Guest switches to kernel state. Signal occurs on the process. The process continues to execute with a different PC (signal handler). This procedure is deterministic therefore we do not need to be exact.

3. An interrupt occurs during a process execution. Guest switches to kernel state. The process is terminated. A new process starts with the same page table base. The new process continues to run (with a different PC, but those two processes are irrelevant and no signal occurred).

The following sequences of events illustrate some important examples of signals and how they fall into the categories we mentioned.

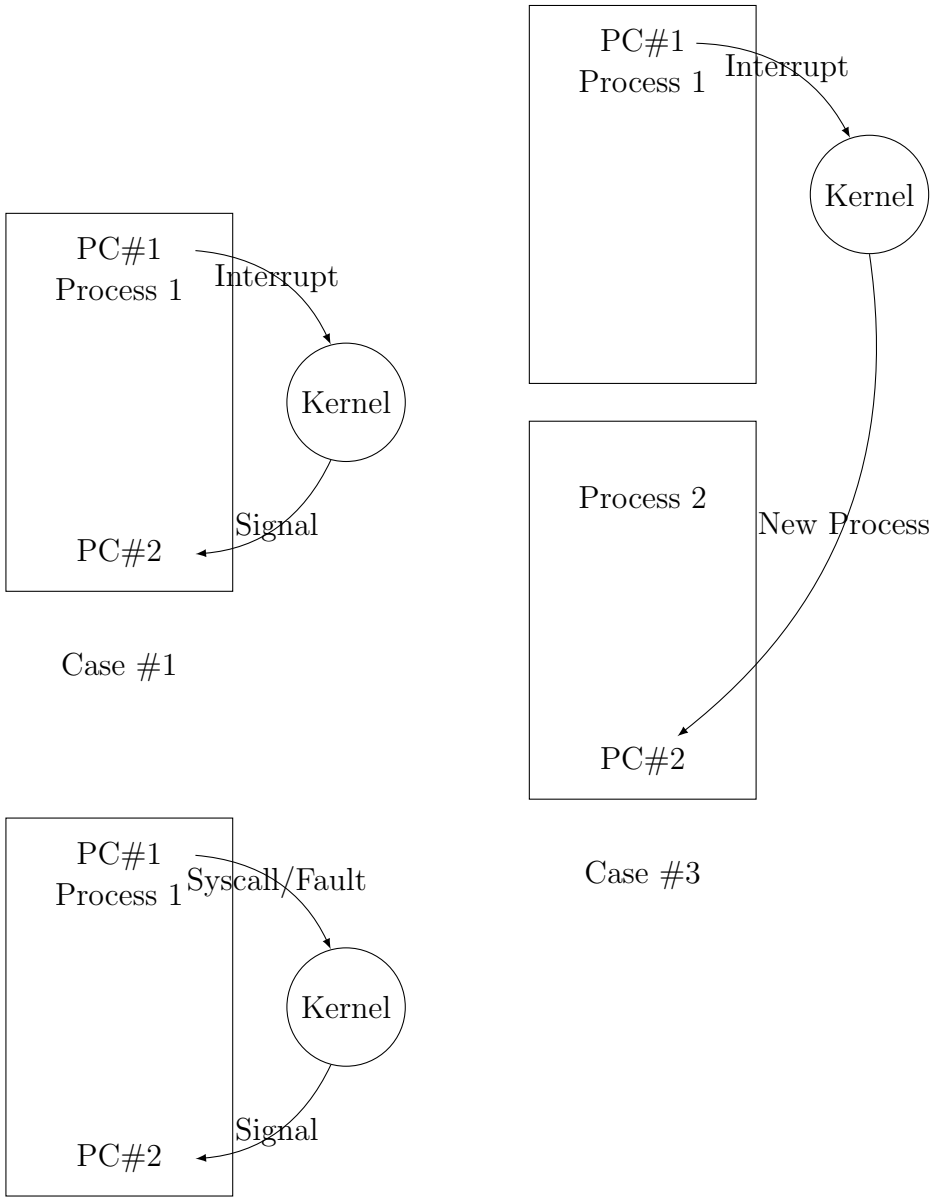

FIGURE 4.1: Signal handling during record and replay

1. An interrupt occurs during process A execution. Guest switches to kernel state, schedules another process B to run. Process B signals Process A. This is an example for the case 1 we mentioned above, which requires a precise point of injection for the first interrupt with respect to process A.

2. An interrupt occurs while process A executes. It switches to kernel state and schedules process B. Process B does a system call into kernel. While in B's context, an interrupt triggers causing the kernel signaling process A. Process A is scheduled and execution continues on a different PC. This is an example for the case 1 and case 2 we mentioned above. The first interrupt requires a precise point of injection with respect to process A, however the second interrupt does not require any special handling due to process B syscalling into the kernel to transition into kernel context.

The following algorithm guarantees detection of signals in these cases:

```
Maintain:
    <Dictionary1>(PageTableBase, usermodePC)
    <Dictionary2>(PageTableBase, guest interrupt)
On guest deterministic entries (Faults, Syscalls) into kernel mode:
    <Dictionary1>.RemoveKey(PageTableBase)
On guest interrupts:
    <Dictionary1>.(PageTableBase) = usermodePC
    <Dictionary2>.(PageTableBase) = guest interrupt
On guest return to user mode:
    oldPC = <Dictionary1>.(PageTableBase)
    if (targetPC != oldPC)
        Intr = <Dictionary2>.(PageTableBase)
        Intr.MarkAsRequirePreciseInjection()
```

Note that case 3 mentioned above will produce false positives under our algorithm. Due to the rarity of the case and no negative impact with false positives (we provide a precise point of injection even though a precise injection is not required), we don't need to filter them out.

Shared memory requires the same treatment due to memory operation sequencing. The essence of our approach treats each user process and the kernel as isolated entities that usually do not affect each other in a non-deterministic manner. Sharing memory between user processes or user process `mmap` into kernel space breaks such isolation. Therefore, it is important that, for example, if an interrupt happens between process 1 writing to shared memory variable A and process 2 writing to the same place, that exact ordering is respected. Shared memory can be detected when two user processes have pages tables that maps to the same page with user mode writable privileges in anyy mapping, or when the kernel tries to access pages that are mapped into a user process during interrupt processing.

In either of these two cases when a precise point of injection is required, we use an optimized hardware breakpoint handler to check for injection conditions. During the recording, we record the register values and machine memory hash at the point of injection. When we replay the trace and an exact point of injection is required for a specific interrupt, we place a breakpoint on the PC of the point of injection, and on each occurrence of the breakpoint exception, we check that the register values and machine memory hash match before allowing the injection to happen.

While our implementation of record and replay is useful and logically correct, there are a few limitations to this approach. First, we cannot support multiprocessor record and replay. Second, we take some performance hit if a user process tries to use hardware floating point unit (FPU). The reason behind this is that the OS usually uses lazy hardware FPU save and restore, which, in our approach, can cause incorrect sequences of events during replay because we rearranged the instructions

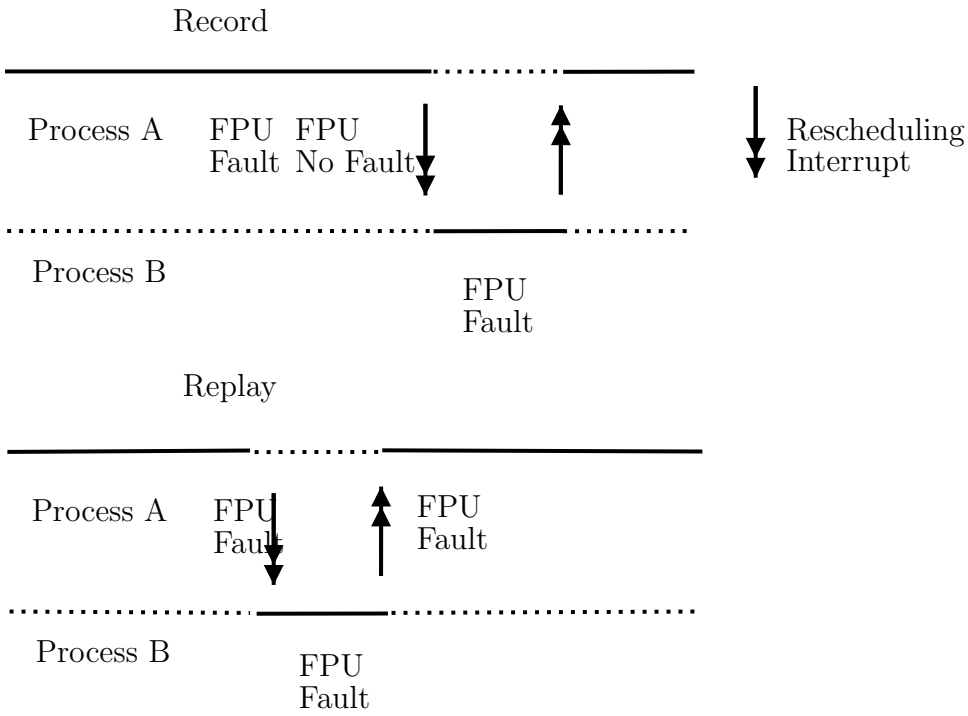

FIGURE 4.2: FPU does not work with interrupt reordering

and generated extra FPU lazy save/restore faults (Figure 4.2). Hardware floating point instructions that do not fault during recording can fault during replay due to this rearrangement. Therefore, in order to avoid this issue, if a process starts to use hardware floating point, instead of lazy save/restore, a full save/restore must be performed thereafter for that particular process, resulting in slightly increased context switch latency for those processes. This increased overhead, however, is less of an issue in modern processors since most processes use FPU and OSes don't always lazy save/restore.

In summary, we have described how record and replay is traditionally done and why such a hardware assisted approach is not available on ARM. We proposed a software solution and claim that in many cases a precise point of injection is not necessary, and when required, optimized hardware breakpoint handling can be used to check for injection conditions.

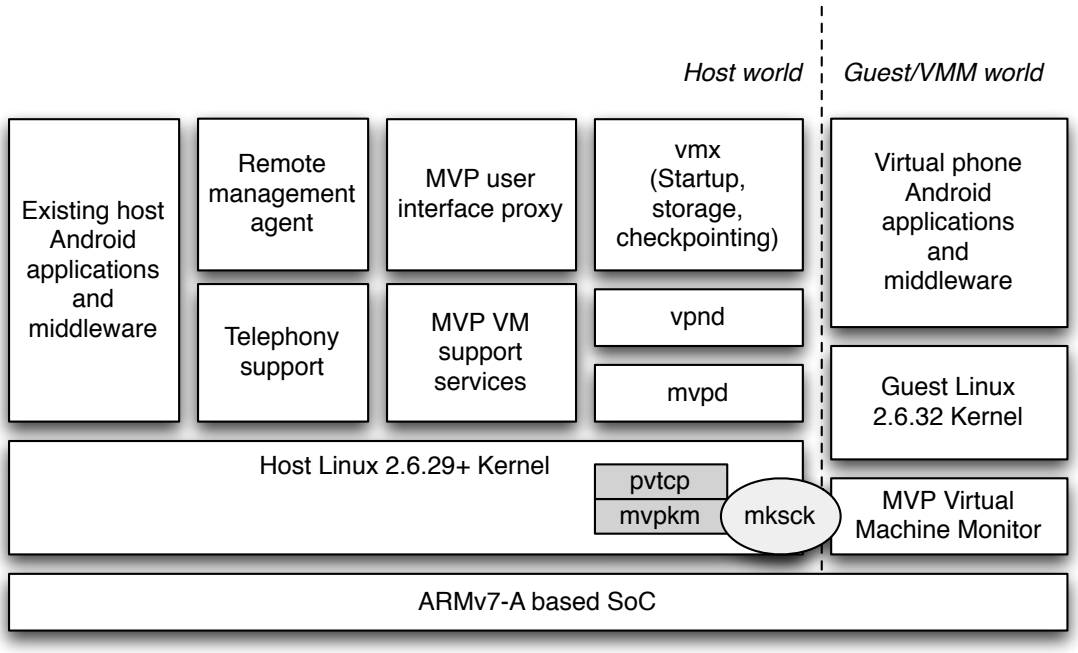

FIGURE 4.3: MVP system architecture

4.4 Implementation

We implement the above record and replay techniques on VMware′s MVP (Mobile Virtualization Platform) (Barr et al. (2010)).

Figure 4.3 shows the architecture of MVP. MVP is a type 2 hypervisor designed for ARMv7. The guest OS is a paravirtualized Linux that is directly executed under user mode and is under the control of the VMM. The host can be any version of Linux typically found in smartphones, e.g. Android. The host processes and OS run just as they would normally, and a dedicated VM thread initiates the execution of VMM and the guest by world switches, where the VMM/guest page tables, interrupt vectors and coprocessors are loaded. The VMM encapsulates the guest and controls the virtualization environment by maintaining a guest privileged mode and a guest user mode, identical to the two modes a normal processor has for an operating system running on bare metal. The paravirtualized guest calls into the VMM using

hypercalls for sensitive privileged operations that cannot be performed normally under user mode, such as switching between privileged mode and user mode, disabling interrupts and performing I/O.

For the purpose of our implementation, in order to maintain logical correctness during replay, we need to make modifications to the guest to ensure deterministic guest kernel behavior. Other than fixing the interrupt injection point in the kernel, which we will discuss later, we need to modify how calibration of the delay loop is done. This procedure executes an infinite loop on the CPU until a few timer interrupts occur. It then calculates, on average, how many loops the CPU can execute for each period of timer tick (i.e. loops per jiffies, a.k.a. LPJ). This value is then used by the kernel to simulate busy loop delays. This calibration procedure generates unwanted, non-deterministic kernel behavior because it requires injection of interrupts at non-fixed points. For the purpose of our implementation, we pass a fixed LPJ from the VMM to maintain the same behavior between record and replay.

The guest kernel directly interacts with the VMM using hyercalls to request CPU and memory services such as interrupts and paging. The guest uses hypercalls, for example, to query the status of the interrupt completion, and to set up page tables for user processes. The VMM communicates with the VMX on the host to submit timer, storage and networking requests via asynchronous socket communication. Upon completion of the these requests, the VMX signals the VMM about the completion and the VMM delivers an interrupt to the guest just as a normal CPU would do, i.e. banking the user mode registers and jumping to the PC of the interrupt handler. In order to support our record and replay implementation, we modify the VMM so that when the guest is in kernel mode, it only delivers interrupts at the preemption point when the guest calls WFI (which is a special hypercall) and refrains from delivering any interrupts for any other hypercalls, so that we maintain one fixed point of injection for kernel events. In contrast, if an interrupt occurs in the guest

user mode, the event is immediately delivered. When an interrupt is delivered to the guest, we record the event and additional information that enables the replay to identify its point of injection. In order for the replay to correctly identify the point of injection for a non precise interrupt, we need:

1. Number of return to user mode hypercalls executed since guest start
2. Number of hypercalls executed since guest start
3. Number of faults delivered since guest start
4. Number of syscalls executed since guest start

In theory we could combine the four counters to form a logical guest event clock indicating guest progress. However we kept separate counters in our implementation for ease of debugging. It is important to note that the count for the number of faults delivered should only count those faults destined for the guest (i.e. VMM cannot handle it), and not include hidden faults such as shadow page table faults that are resolved by the VMM, because those faults are non-deterministic and guest transparent.

In order to ensure a correct replay, all hypercalls that return information on any state of the guest VM must be recorded. They include timer hypercalls to get current tick and time of the day, and hypercalls to query I/O completion status.

To illustrate what information is required for record and replay on MVP, we describe how disk I/O is correctly recorded and replayed. Figure 5.7 shows the architecture of the virtual disk device in MVP. An I/O request starts with a guest kernel hypercall into the VMM requesting a block read. The result of this request (i.e. return value of the hypercall) must be recorded. Then the request is passed over the VMX side, which in turn interprets and submits the request to the host operating system. Later, the host notifies the VMX about the completion of the

request, which in turn notifies the guest by triggering an interrupt in the guest OS. Since this event is non-deterministic (depending on the speed of read of the host), we record the point of injection of this interrupt using the guest logical event clock. The guest may, upon receiving the interrupt, query the result status of the I/O request in the form of a hypercall, whose return value must be recorded. During replay, the same sequence of events should occur, with the guest OS submitting the request first. At that point, we simply reply with the same value as we recorded, and carry out the request in the host OS. Assuming the same start state of the VM virtual disk between record and replay, the I/O request is going to return with exactly the same result however with a different delay from the host performing the I/O. The VMM then monitors the guest event counters to reach the same number of events required to inject the interrupt for I/O completion, and examines the status of the I/O. It might have already occurred, in which case we simply inject it for real, or it might not have, in which case we have to halt the guest and wait for the host to complete the request. Then, we simply inject the interrupt, after which the guest should query I/O completion status with the same hypercall, which we return with the value as recorded. Note that we do not need to record the I/O blocks and simply rely on the fact that the virtual disks will be at the same starting state between record and replay, and therefore, with deterministic replay, should arrive at the same state and transfer the same data between record and replay. Moreover, interrupt arrivals are "throttled" during replay, and will not trigger a guest interrupt immediately until the guest executes to the point of injection.

In order to support precise point of injection during replay, we need to detect when precise point of injection is required for an interrupt. We maintain *process affiliation tuples*, a set of tuples in the format of *(Guest Page Table Base, PC, Interrupt)*. This tuple is updated whenever an interrupt occurs using the current process′s page table, PC and the involved interrupt. We remove a process from the set when a process

does a system call or faults, because in those cases the process becomes deterministic and is "off the watch list". When a "return to user" hypercall occurs which usually signals the end of interrupt processing and returning to user mode, we check the guest user mode PC against the set of tuples to discover possible signals. If the PC we are returning to is different from the PC we recorded previously, a signal might have occured and the interrupt involved requires a precise point of injection.

Another case that requires a precise point of injection during replay is shared memory. We detect this case by enforcing a memory mapping policy that any physical page may only have one user writable mapping at any time. The shadow page table maintains a backmap that records all active mappings for a specific physical page. This back mapping is updated each time a new mapping is created. Whenever a new user mode mapping is created, we check that the new mapping is the only user mode mapping for the physical page if the any previous user writable mapping exists, and if that is not true, for all the previous mappings, we check against the process affiliation tuples using the page table base to detect any registered interrupts. If detected, it means that another process was trying to access the same page we are about to access using a user writable privilege, and a precise point of injection is required for that interrupt because shared memory occurred.

In order to ensure we inject at the correct place during replay for precise points of injection, we need to record the register contents and memory hash at the point of interrupt for comparison during replay. Each time an interrupt occurs during recording, we need to compute the memory hash for the specific process the interrupt occurred in. Obviously, it is inefficient to compute for every single page of the process each time an interrupt occurs. Instead, we write protect the entire guest memory and mark a page changed if the write protection is broken by the guest. Each time when an interrupt occurs, we only compute the hash for the broken pages and re-protect them, and we add the total hash with the hashes computed from the broken

pages. An implication of this approach requires that the hash is additive. During replay, we only compute the hash when a precise point of injection is required. For our implementation, we use XOR as the hashing function.

4.5 Experiments and discussions

Compared to original MVP implementation with record and replay implemented, our approach creates extra overhead in two areas. The first one is the overhead of overlaying the event recording mechanisms and precise interrupt injection detection on top of the current MVP implementation. The second one is the overhead of performing memory hashing. We are concerned with how our implementation would perform for kernel-intensive benchmarks, user-intensive benchmarks and real life applications. We would like to evaluate how much overhead is generated in each of these two categories, for each type of workloads.

To evaluate our implementation, we run several micro-benchmarks and some application benchmarks to validate the correctness of our approach and to show the overhead of our design. We evaluate our work running MVP on a Pandaboard. It has an OMAP4 ARMv7 processor with 1GB of memory. The guest virtual machine disk images are stored on a RAM disk for all experiments. The reason for doing so is because the default storage system, a SD card, is very slow and can take roughly 80% of the total time of the benchmark run-time if the benchmark is I/O intensive, overshadowing the true overhead of our implementation. By running all experiments on RAM disk we ensure that the CPU is sufficiently utilized at all times and the overhead we measure reflects a conservative view of the implementation's overhead. We also made sure that no other host processes were running, and dynamic power/frequency scaling were off. For each experiment we run 5 trials and calculate the average and standard deviation of the result. For each trial, we reset the VM image to a clean image we created. Table 4.1 lists how each experiments work.

Table 4.1: Evaluated experiments

Experiment name	Experiment detail
memcopy	Copy 64MB of page aligned memory data 100 times.
fork	Fork and execute a near-NOP program 1000 times.
getpid	`getpid()` syscall 3500000 times.
getuid	`getuid()` syscall 3500000 times.
memory read	Create a circular linkned list spanning 64MB and read through 10000k nodes.
process switch	Create two processes and ping pong between them 1000000 times.
thread switch	Create two threads and ping pong between them 100000 times.
process create and destroy	Measures the time to create and destroy a new child process 2000 times.
segv	Triggers `SIGSEGV` 1500000 times.
nsort	Sort an array of long intergers.
ssort	Sort an array of strings.
bitfield	Bit manipulation functions.
fpemu	Software floating point.
fourier	Calculate fourier transform coefficients.
assignment	Task allocation algorithm.
huffman	Compute Huffman encoding.
idea	International Data Encryption Algorithm.
nnet	Back-propogation network simulation.
ludecomp	LU decomposition test.
busybox compile	Compile busybox 1.21 from scratch.
kernel compile	Compile Linux kernel 3.4 for ARM.
Python computation	Compute N-body problem using Python.
Python threading	Spawn and bounce among 20 Python threads.

Figure 4.4 shows the experiment results for the micro benchmarks. The result shows that we typically have an overhead of 20% to 85% during recording and replay. Replays are typically slightly faster than recording due to not having to wait in real time for timers. Another possible cause for the speed increase is that exceptions may coalesce during replay, resulting in better cache performance. The process switching and thread switching benchmarks have higher overhead due to the way the benchmarks are implemented. These two benchmarks are implemented using

`fork()` which makes them share memory. Our implementation causes extra memory faults due to processes not sharing the same user writable mapping. As the processes switch from one to the other, they trigger extra repeated hidden faults (Figure 4.5). In addition, we observe significant overhead on memory write benchmarks due to having to compute the memory hash on more pages than for other micro-benchmarks (Figure 4.6). These experiments show that the majority of the overhead comes from two aspects of the implementation. The first one is the extra faults generated by shared memory detections, because we hardly generate any overhead if we are only dealing with processes without shared memory as shown in benchmark results other than process/thread switch. The other source of major overhead comes from the cost of hashing the memory as shown in the `memcopy` experiment, because we do not generate the same huge overhead if we are just reading the memory as shown in the `memory read` experiment.

In general, many of the micro benchmarks test virtualization bottlenecks, such as system calls and MMU operations, and show the worst case performance for each of the categories we test. On the contrary, Figure 4.7 shows the experimental results for `nbench`, which is a purely computational benchmark. It is a well known benchmark for testing only the computation power of the CPU. The details of each category of the benchmark is also listed in Table 4.1. In these experiments, our implementation hardly generates any extra overhead for CPU bound workloads as expected. Note that we do not have a replay result for this suite of benchmark because this benchmark consists of score based tests that run for a fixed amount of time. The replay result would only show that we have a same benchmark result as recorded because any `gettimeoftheday` result of the replay would be the same as recorded.

Figure 4.8 shows the experimental results for two of the worst case scenario for replay that could occur in our implementation: a specifically designed empty loop,

that has no interaction with the kernel, and triggers signal/shared memory once in a while:

```
//signal test
sigaction(SIGALARM, handler);
set_timer(ONE_SECOND);
while(true) {
    i++;
}
handler()
{
    print(i);
}

//shared memory test
fork();
if (process_id() == parent) {
    while(true) {
        i++;
    }
} else {
    while (true) {
        sleep(1);
        print(i);
    }
}
```

These two experiments cap our worst case overhead for replay at about 20 times. In these two cases, we have to take billions of breakpoint exceptions before reaching

the correct point of injection. However, in reality, this almost never happens, because most programs would do useful work and perhaps do system calls in loops. If a precise point of injection lies within a loop, doing some useful work instead of an empty loop greatly reduces the number of times we hit the same PC during replay. System calls, on the other hand, provide vital determining points to the replay so that it does not have to take multiple faults within a loop at all.

Figure 4.9 shows the experimental results for some applications. GCC induces a high overhead when memory is hashing is used due to the workload consuming a large amount of memory and triggering many hash computations. In contrast, our Python benchmarks have a relatively small active working set and cause roughly 10% overhead. In order to verify that we are indeed nowhere near the worst case scenario we mentioned earlier for real life applications, we measure the average number of breakpoint exception we took for each precise point of injection before a match (Figure 4.10). This number has a high variance because the number entirely depends on how many iterations in a loop has passed when the interrupt triggered. For real life workloads, we have not observed any application that would cause the same tremendous overhead as the signal test and shared memory test micro-benchmarks we mentioned above.

In addition, in order to better understand the source of the overhead, we measure the number of hashes computed for each of the workload we tested (Figure 4.6) and the number of precise point of injection required per second for each workload (Figure 4.11). These two experiments demonstrate that our approach is much more sensitive to active working set size than to number of precise point of injections.

Overall, we are able to maintain a relatively low, 10% to 20% overhead for programs with small active working sets. For programs consuming a large amount of memory, our overhead swings between 1.5 times to a maximum of 3 times.

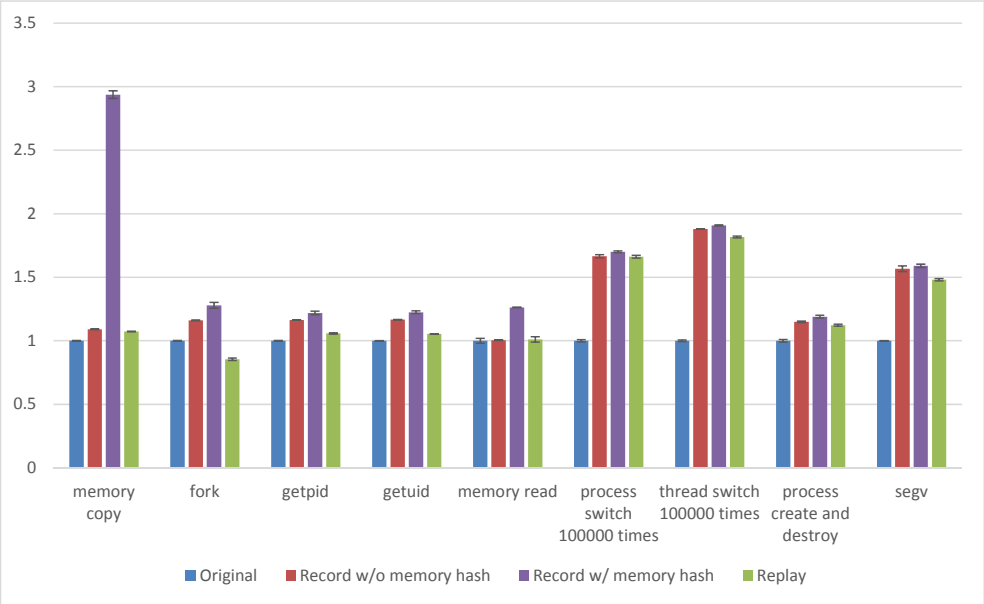

FIGURE 4.4: Micro-benchmarks result (time to completion, lower is better)

4.6 Conclusion

In this chapter, we discussed how record and replay was traditionally done on the x86 architecture and why it is difficult to implement on ARM architecture. Instead of ensuring literally correct replay as was done on x86, we proposed an alternative implementation without any hardware assistance on paravirtualized guests to guarantee logical correctness of the replay. We modify the VMM to be guest-process aware and maintain important information that assists in detecting and handling corner cases of our implementation to ensure replay correctness. We verified and tested our implementation on various micro and application benchmarks and explained our findings on implementation details and record and replay overhead.

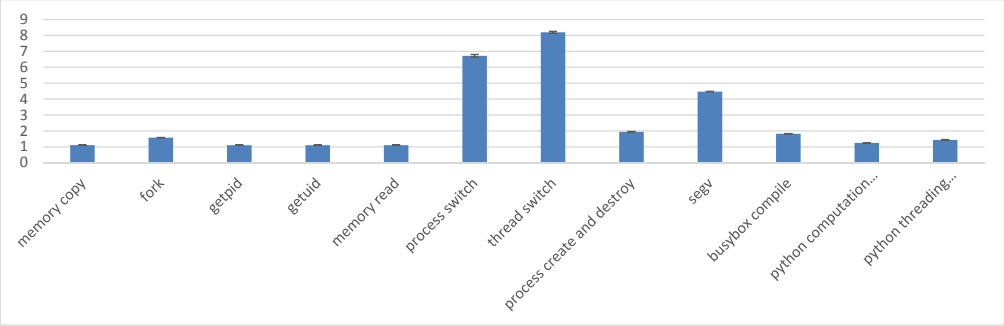

FIGURE 4.5: Number of hidden page faults taken for our R/R implementation compared to original MVP

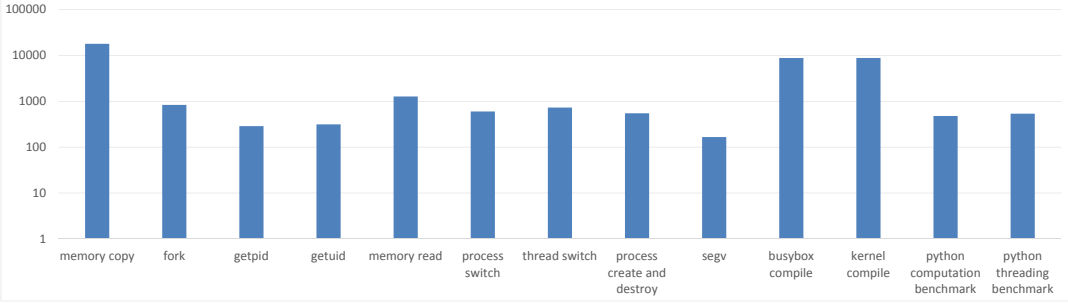

FIGURE 4.6: Number of pages hashed per second during recording

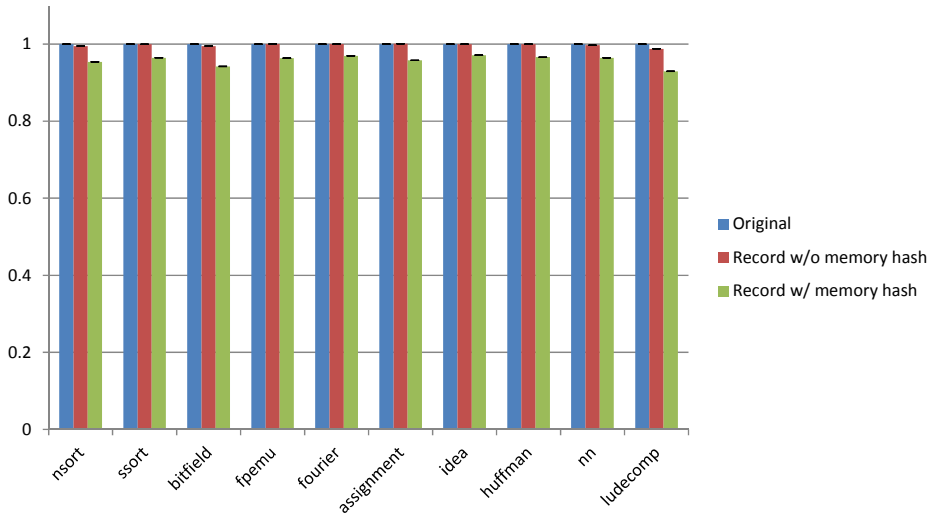

FIGURE 4.7: NBench result (benchmark rating, higher is better)

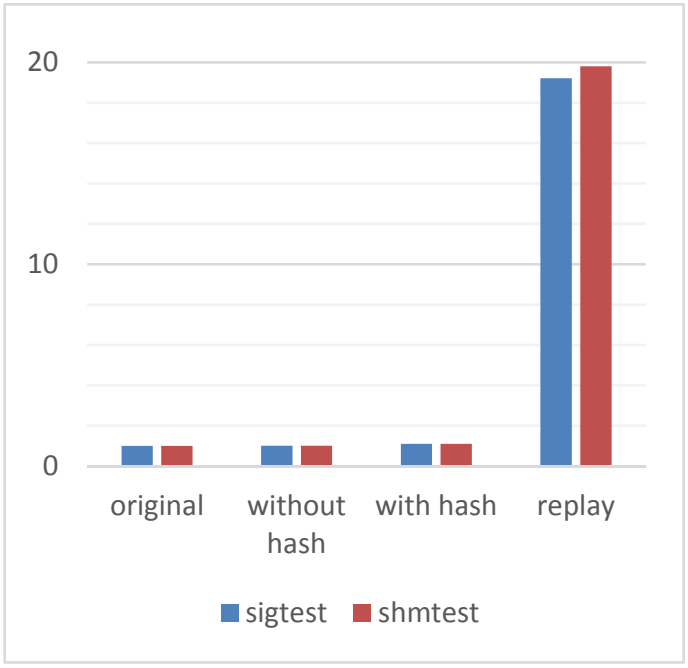

FIGURE 4.8: Extreme micro-benchmarks result (time to completion, lower is better)

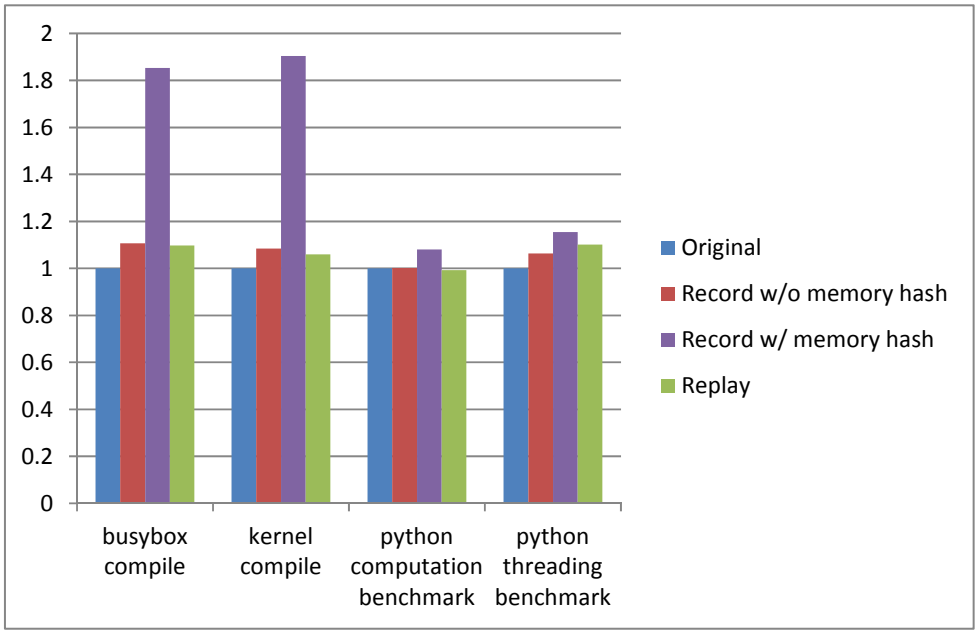

FIGURE 4.9: Application benchmarks result (time to completion, lower is better)

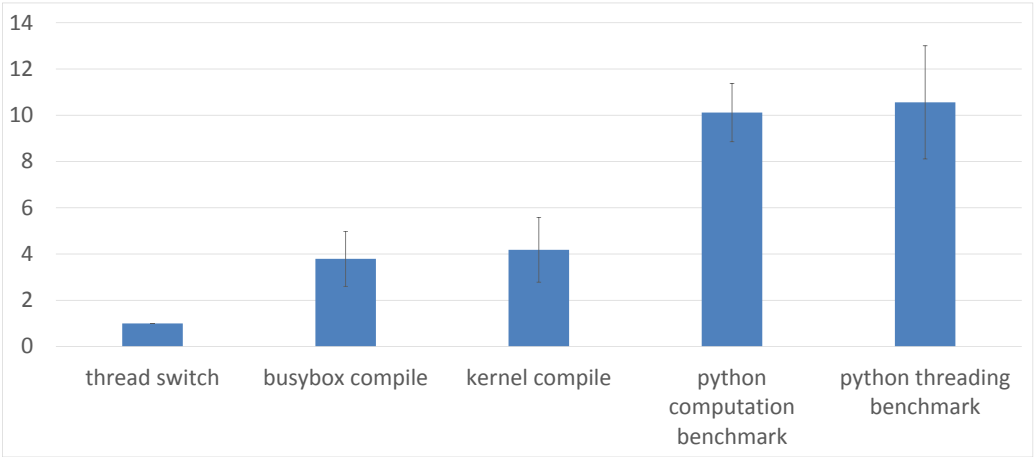

FIGURE 4.10: Average number of exceptions taken before reaching a precise point of injection

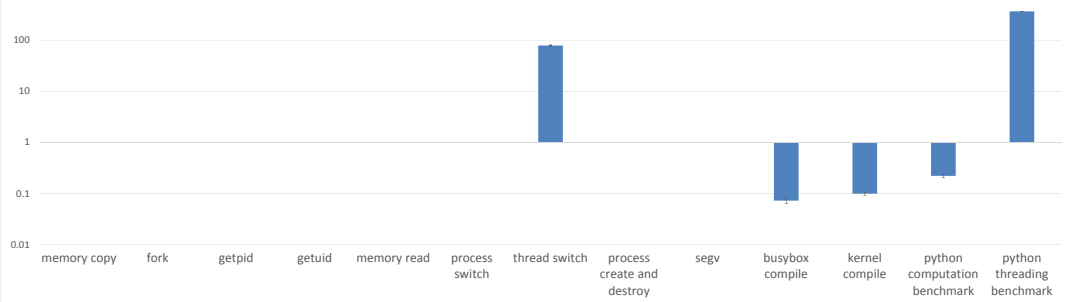

FIGURE 4.11: Number of precise interrupt injection required per second

5

Storage virtualization using logging block store (LBS) on Secure Digital cards

5.1 Introduction

Virtualization on mobile devices introduces several challenges in terms of storage virtualization. The business model of mobile virtualization products encourages that type 2 hypervisors are used so that the smartphone development pipeline is minimally perturbed in trying to satisfy the multi-workspace scenario (Barr et al. (2010)). Therefore, the guest images are stored as an image file on the host file system. However, the device is susceptible to loss of power at any time, which necessitates a robust image file format that can withstand such sudden interruption during guest operation. In addition, typical mobile devices are usually equipped with very limited internal NAND storage and external Secure Digital (SD) card. Because SD cards are usually optimized for FAT file system and sequential reads and writes, the characteristics of the card mismatches with the workload of the virtual machine, which tend to have highly non-sequential I/O mixtures.

In this chapter, we first discuss the characteristics of the SD card, and using

Mobile Virtualization Platform (MVP) as an example, illustrate why the workload of the virtual machine mismatches with the characteristics of the SD card. We then propose a solution using a new logging block store (LBS) image format and discuss its performance gains and various implementation challenges.

5.2 Performance characteristics of SD cards

The design of an SD card optimizes for cost instead of speed. The NAND devices inside an SD card are organized and managed by an Flash Translation Layer (FTL) to give the impression of contiguous logical blocks. Meanwhile, the underlying mechanism usually groups large chunk of NAND devices into erase blocks, the smallest unit which writes can occur. Unlike a solid state disk, due to space and cost constraint on the SD card, the FTL is usually simple and optimized solely for sequential writes.

SD cards has a class rating that reflects sequential I/O bandwidth. This rating, however, does not apply to random writes at all. Random writes are usually very slow compared to sequential writes. Figure 5.1 shows the performance comparison of sequential and random read/write on a 8GB Class 6 ADATA SD card evaluated by `sdperf`, which opens a file or raw block devices and performs I/O benchmarks according to the setup. We observed that there's little difference in sequential and random read speeds but huge distinction between sequential and random writes. Figure 5.2 shows the ratio of sequential to random write bandwidth on five SD cards (Table 5.1) from different manufacturers with different speed class ratings. Other than the outlier of the 8GB SanDisk card which was labeled as Windows Phone 7 compliant (potentially optimized for certain random read/write), the five cards behaves similarly beyond 16KB block size.

Because of the simplicity of the FTL on SD cards, writes usually cannot occur on a small block size. If we issue a 4KB write to the SD card, the FTL has to read a

Manufacturer	Capacity	Class	Alloc. unit	fat cluster
SanDisk^{TM}	4GB	4	4MB	32KB
SanDisk^{TM}(WP7)	8GB	4	4MB	32KB
Kingston^{TM}	4GB	4	4MB	32KB
ADATA^{TM}	8GB	6	4MB	32KB
PNY^{TM}	16GB	10	4MB	32KB

Table 5.1: SD card details.

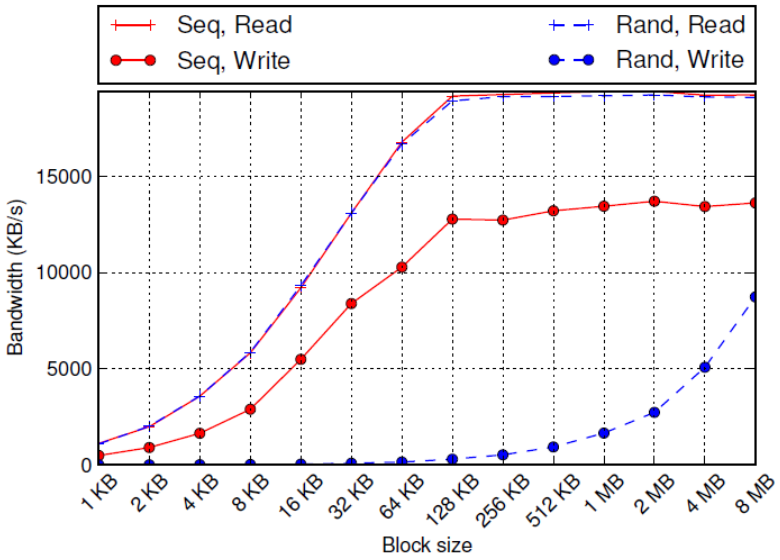

FIGURE 5.1: 8GB ADATA Class 6 SD card I/O bandwidth as a function of block size and I/O ordering

much larger block size called an *allocation unit* (4MB for the card in Figure 5.1), and perform a read-modify-write operation. Some FTL supports efficient interleaving of writes to multiple AUs as long as sequentiality is maintained within each AU (Nath and Gibbons (2010)). Figure 5.3 shows the effect of interleaved writes to multiple AUs. Some cards show good performance up to 4 AUs but the Kingston card only supports one AU. For our work in this chapter, we disregard this potential optimization and assume that non-sequential writes have a high penalty because we are trying to find a uniform solution for all SD cards.

Another point to take note is that a few random writes can drag the performance of the entire I/O workload down to the level of pure random writes. Figure 5.4

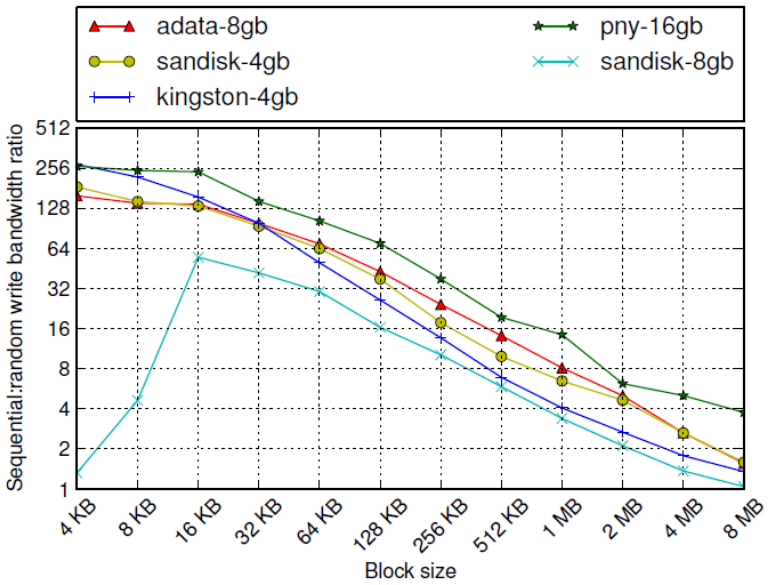

FIGURE 5.2: Sequential:random write bandwidth ratio as a function of block size

shows an example, where we randomly insert random writes as a percentage of the workload. Even 10% random writes slows down the entire workload to the level of pure random writes.

Lastly, the I/O performance of the SD card can also be affected by the state of the file system on the card. The FAT file system can fragment and degrade over time, therefore causing reads and writes to a single file to be as slow as random reads and writes. It matters then, when we try to store virtual machine disk images on a FAT formatted file system because we must pay attention to avoid fragmentation when allocating such images on the SD card.

5.3 Characteristics of the virtual machine I/O

From the perspective of the host, the virtual machine generates mostly non-sequential I/O. There are three factors that contribute to the I/O characteristics:

The guest uses FAT and ext3 partitions. FAT is naturally fragmentation-prone and ext3 journaling generates non-sequential writes. Figure 5.5 shows a write trace

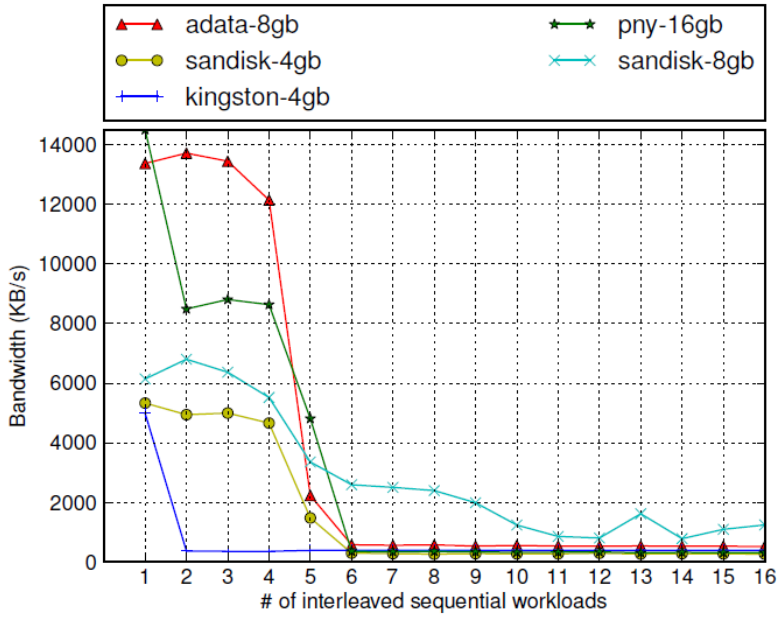

FIGURE 5.3: Write bandwidth as a function of the number of interleaved sequential workloads, separated by 2AU, at 256KB block size

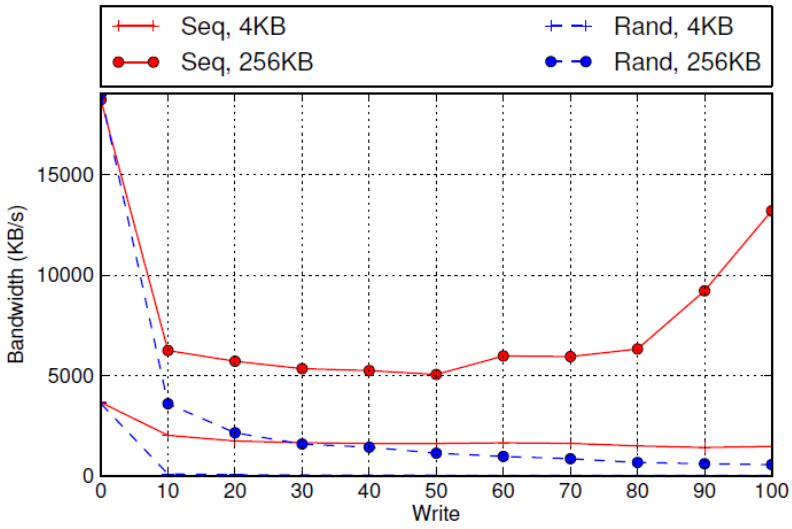

FIGURE 5.4: 8GB ADATA Class 6 SD card I/O bandwidth as a function of write percentage in I/O mixture and I/O ordering

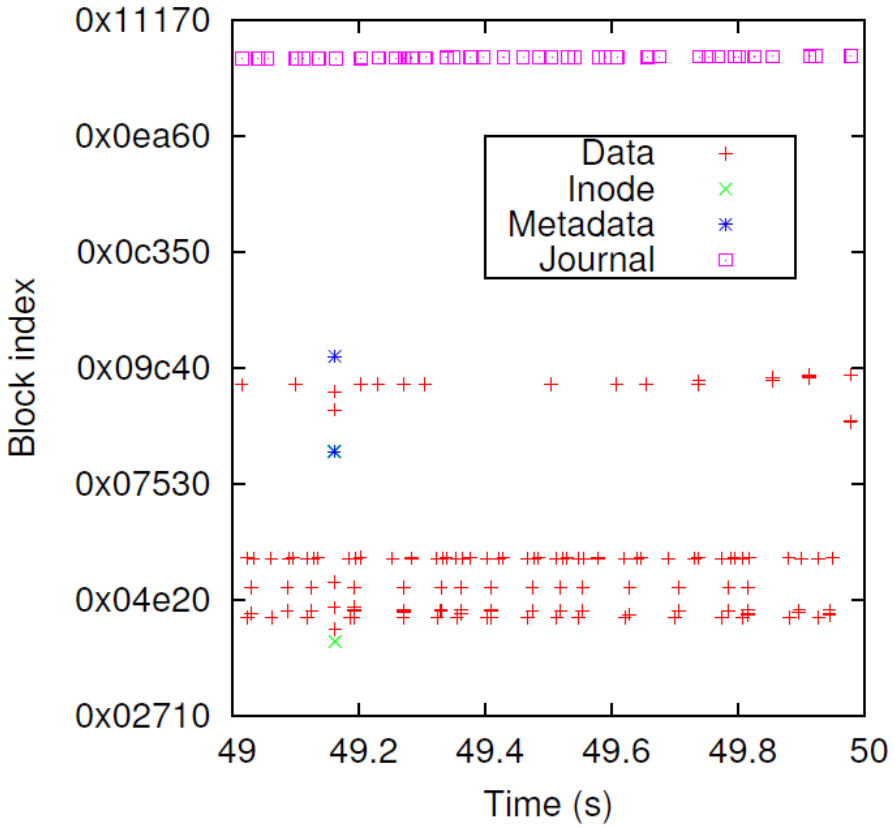

FIGURE 5.5: 1 second sample of browsing session writes on ext3

to an ext3 partition during an Android 2.2 web browsing session. In addition to the web browser accessing non-sequential browser data on the partition, the access generates additional non-sequential writes to the meta-data, inodes and journal data on the ext3 file system.

In order to provide robustness to the guest which we will discuss later, the guest frequently checkpoints memory and CPU states so that it can recover from last session or loss of power gracefully. An adapted Clock-Pro (Jiang et al. (2005)) working set estimation algorithm is used to selectively write cold pages to persistent storage periodically so that it is faster to perform the checkpoint when one is actually requested. This procedure generates non-sequential writes as shown in Figure 5.6.

Many applications in the guest generates non-sequential writes while writing to configuration or temporary files. We examine several applications' I/O traces:

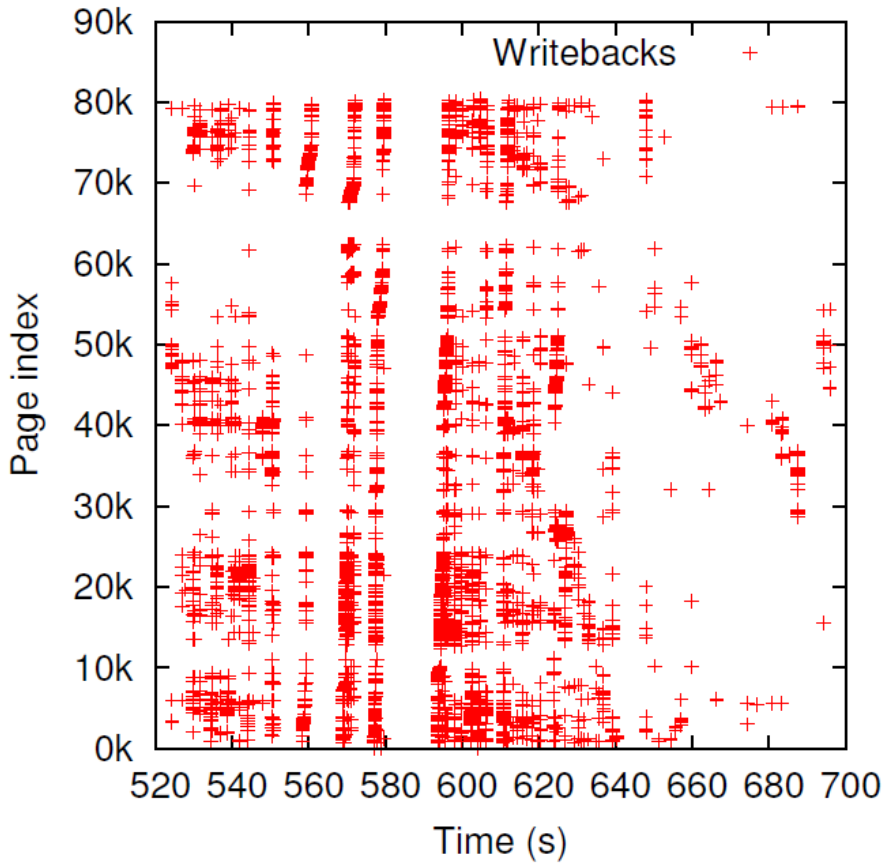

FIGURE 5.6: 180 second sample of background cold page writebacks of a large space of guest physical memory

1. Android boot. The initial boot of an Android OS until the `BOOT_COMPLETED` intent.

2. Contacts Database. Import 2000 contacts into the Android Contacts application and then search for and delete 40 contacts.

3. Mail Client. Use Android Mail 2.2.1 to access a mailbox of 24MB with 356 messages and 3 folders. The length of the messages and the attachments were generated by SPECmail2009 benchmark.

4. Slideshow. Browse through 52 NASA images using Astro file browser.

5. Web Browsing. A one second sample of web browsing using Android 2.2

Name	Writable	Filesystem	Description
`/system`		squashfs	Android binaries
`/data`	x	ext	user-installed programs and data
`/cache`	x	ext	cache space used by Android
`/sdcard`	x	fat	SD card for multimedia files
`/flex`		squashfs	enterprise customizations

Table 5.2: Disk partitions in the Android guest under test.

browser.

Table 5.2 shows the partition layout in our experiments. Note that squanshfs partitions are not writable. Table shows the result of our experiments. The size of the trace is expressed in KB and as a count of total operations. A rough classification of I/O sizes is provided in which each column is exclusive of adjacent columns. "Skip" is a measure of sequentiality: the number of block accesses that are not adjacent to a previous access. A skip percentage of 100 represents a completely non-sequential workload; a skip percentage of 0 is completely sequential. Write and barrier percentages are relative to the total number of I/O operations per partition. The results demonstrated that small writes dominates the I/O profile for some workloads such as Contacts Database and Email. In Section 5.2 we already established the fact that random workloads only need 10% writes to be as slow as pure random writes. Therefore, the I/O characteristics for many applications can be effectively counted as random writes.

Table 5.3: Charateristics of I/O traces (read/write breakdown).

	read						write							
	size	count	I/O Sizes (KB)			skip %	size	count	I/O Sizes (KB)			skip %	write %	barrier %
partition	(KB)		1	≤ 4	> 4		(KB)		1	≤ 4	> 4			
Android Boot														
/system	18562	1275	21	39	1215	0	0	0	0	0	0	NA	0	0
/data	28	24	24	0	0	41	16468	1204	120	43	1041	1	98	2
/cache	11	7	7	0	0	27	18	9	7	1	1	19	56	6
/flex	29	6	3	1	2	7	0	0	0	0	0	NA	0	0
Contacts Database														
/system	763	64	8	6	50	1	0	0	0	0	0	NA	0	0
/data	855	83	32	21	30	4	55281	21048	15414	3348	2286	17	100	4
Email Client														
/system	6218	511	80	26	405	1	0	0	0	0	0	NA	0	0
/data	2473	191	53	51	87	3	52100	16668	10520	2595	3553	12	99	33
/sdcard	6	3	2	1	0	17	112	97	90	7	0	43	97	0
Slideshow														
/system	5664	449	55	23	371	1	0	0	0	0	0	NA	0	0
/data	500	82	27	23	32	8	6452	659	173	73	413	2	89	10
/sdcard	38	18	14	2	2	22	13701	1883	1407	197	279	6	99	0
Browsing														
/data	43895	1662	313	228	1121	1	12609	3512	2149	542	821	10	68	18

5.4 Logging block store (LBS)

The disparity between the performance characteristics of the SD card and the characteristics of the virtual machine I/O motivates that we come up with a solution to bridge the gap. In addition to the problems aforementioned, we also need to ensure that the virtual machine storage system is secure and reliable. Because of the size limitation of the internal NAND storage on many mobile devices, the virtual machine images must be stored on external SD cards, which is susceptible to security attacks because the entire card has limited access control and is readable by all applications with extended storage access permission on Android. Also, because of the volatility of the battery power of the mobile devices, we need to be able to deal with sudden loss of power without corrupting the VM images.

In this section, we present our solution: logging block store (LBS). Figure 5.7 illustrates how LBS interacts with the rest of the virtual machine systems. As MVP is a paravirtualization platform, the frontend of the guest communicates directly to the virtual machine's `vmx` storage system via hypercalls. LBS is responsible for interposing on top of the guest requests and translates them into host specific I/O

operations and calls the host API to perform the actual I/O on the physical devices.

An LBS formatted image contains two separate files, a data file (suffixed `.lbsd`) and a meta-data file (suffixed `.lbsm`). The data file contains the actual content of the image, and is stored on an external storage (SD card) due to its significant size. Meanwhile the meta-data file contains structural information and the meta-data of the data file. Because the meta-data file is relatively small in size, we store this file on the internal storage which is usually formatted with JFFS2 or ext3/4 file systems and has access controls, so that the more secure and robust file system on the internal storage can protect the meta-data file and simplify our design. Figure 5.8 and Figure 5.9 illustrates the format of LBS files.

5.4.1 LBS format

The LBS data file is divided into fixed 1KB size *blocks* and fixed 256KB size *clusters*. In order to solve the issue of non-sequential writes from the guest we mentioned before, when the guest writes a block to its storage, from the virtual machine's perspective, it only writes to a logical block represented by a *logical block number* (LBN). The virtual machine then translates this requests into writes of some physical block indexed by *physical block number* (PBN) within the LBS data file. To ensure that we only write sequentially to the physical devices at maximum possible speed, all writes always append to the end of current active cluster and are buffered until the entire cluster is filled up or the guest issues a barrier. LBS maintains a list of empty clusters and activates another empty cluster when the current one is fully written. A cluster can become close to empty when a physical block within the cluster are disassociated with its logical block because the logical block is written again by the guest. A garbage collection (GC) thread in the VMM maintains a list of near empty clusters and reclaim them when the data file is running out of empty clusters.

The LBS meta-data file is an append-only log of meta-data and barrier entries.

Each writes from the guest appends a LBN → PBN mapping entry to the meta-data and each barrier from the guest appends a barrier entry to the meta-data. Each meta-data entry flags and optimizes for zero blocks and also has checksums and timestamps for non-zero blocks. When a meta-data file is opened or reaches maximum size, we perform a sweep of garbage collections to remove stale entries.

To minimize latency, the data file is opened using `O_DIRECT` flag to bypass any host buffering and the entire data file is fully allocated on the FAT partition once created, avoiding any further fragmentation issues. The meta-data file requires about 12MB per 1GB of logical block space. It is stored on the internal storage and opened with `mmap` to achieve both speed and space advantages by letting the host manage the caching and writebacks of the meta-data file according to its memory pressure.

5.4.2 Reliability and security

The reliability of the LBS file system is based on the robustness of internal storage where the meta-data file is stored. The internal storage is usually formatted with YAFFS or JFFS2, which guarantees the robustness of the meta-data file. LBS cannot provide those reliability guarantees when the meta-data file is not stored on a log structured file system. In addition to the inherent file system protection on the meta-data file, when guest issues a barrier write, we checksum all meta-data entries since last barrier and preserve that information with the barrier entry we write to the meta-data file. If a missing or corrupted barrier is detected, we rollback to the last barrier which maintains the expected guest barrier semantics.

When the meta-data file is sufficiently robust, the block checksums in the meta-data file, which are computed by 32 bit Fletcher checksum (Fletcher (1982)) or SHA-256 checksum, guarantees that the data file is robust, as long as the data file is synced to the physical media before the meta-data file is. The checksums are also able to detect media failures as well as tampering or corruption by attackers or other

applications.

There exists two major security threats for the storage system: attacks by software from within the phone, and attacks from loss of physical device. We encrypt the LBS data file at block granularity with XTS-AES cipher and store the key in the internal storage. We have the same security guarantee as the Android application keystore in the event of loss of physical devices. We also protect the data file from being maliciously modified by other software on the device. However, encryption alone does not protect the data file against replay attacks which attempts to inject a known encrypted block from the past into the LBS data file. This is prevented by introducing a logically clocked timestamp for each block that increments after each write and have this timestamp involved in the block checksum process. In this way, if a replay attack occurs, the wrong timestamp would cause the checksum to mismatch.

5.4.3 Garbage collection

LBS system spawns a separate thread for each image that garbage collects clusters in the background. A cluster is collected when and only when the number of free clusters is low. The garbage collection algorithm selects clusters based on a number of parameters. A collected cluster will have its occupied blocks loaded into the write queue in the background and later written to the current active cluster. The LBS data file is over-provisioned (e.g. 112%) to ensure that there are adequate free space to optimally run the GC.

The GC selection should take many factors into account. A simple and naive algorithm only takes the "emptiness" of a cluster into account. In reality, this algorithm is often insufficient and slow. Because the geometry of the SD card can be quite different and affected by many factors as discussed in Section 5.2, the LBS data file, when allocated by the FAT file system during creation, can sometimes have frag-

mentation or not start at an AU boundary. This means that a seemingly contiguous write to the LBS data file's clusters can actually span multiple AUs and therefore degrades to non-sequential writes to the physical device when non-contiguous cluster writes occur. For example, a write to cluster 5 then a write to cluster 4 would, in an ideal case, translates to a write to AU 5 then write to AU 4, which would mean a sequential write to both AU. However, misaligned geometry sometimes cause the clusters to not start on AU boundary and the above example would translates to partial writes to AU 5 and 6 then partial writes to AU 4 and 5, which is terrible in terms of performance. In order to solve this issue, we introduce extra weighted parameters in the LBS GC selection algorithm to try to guarantee as many contiguous free clusters as possible:

1. Emptiness: The number of unoccupied blocks within a cluster.

2. Left empty: An award for clusters whose left sibling is empty. Promotes contiguous clusters.

3. Outlier correction: An award for clusters who have very few unoccupied clusters but have siblings that are very empty. We try to treat an outlier within a contiguous near empty clusters the same as if it is almost empty because collecting such full clusters can extend the length of a contiguous series of clusters.

4. Write position: An award for clusters who is next to the current active cluster. This promotes contiguous writes.

Those four components are equally weighted for experiments in this chapter but depends on the amount of over-provision in the data file. The more over-provisioning we set, the more important "write position" and "left empty" become. The less

device ID	the partition accessed by this I/O
R/W	flag indicating whether this I/O is a read or a write
fragment	position in scatter-gather list
offset	location on disk
length	number of bytes accessed
barrier	flag indicating that this record represents a barrier request

Table 5.4: Trace record

over-provisioning we set (below 25%), the higher the GC pressure is, and the more important "emptiness" is.

5.5 Experiments and discussions

In order to evaluate the performance of LBS, we perform several synthetic and trace-based tests to show the benefit of LBS and the cost of write amplification, integrity checking and encryption.

All experiments are done on an HTC Nexus One with 1GHz Qualcomm Snapdragon chipset and 512MB of DRAM. The phone ran CyanogenMod 7.1, an after-market version of Android 2.3.

We use `sdperf` discussed in Section 5.2 to perform the synthetic tests and use `blksim` to perform the trace-base tests. `blksim` replays a trace gathered during execution of a guest application. The format of the trace is shown in Table 5.4. `blksim` replays the trace by first reading the trace into the memory and then opening a raw block device using `O_DIRECT` flag and issuing I/O commands as recorded.

We use the same workload as desribed in Section 4 with the Android Boot and Slideshow workloads repeated five times to create longer runs and reduce variance.

5.5.1 Benefit of LBS

We compare the performance of a LBS formatted guest image against a "flat" guest image which does not buffer I/O and uses a simple one to one mapping of LBN to PBN. Figure 5.10 shows the ratio of bandwidth achieved for LBS image to flat image, running synthetic tests. Garbage collection was not triggered in these experiments.

The graph shows similarity between sequential and random reads as expected. The slowdown of LBS is due to the extra layer of abstraction, encryption, and integrity checking. This result suggests that it is better to choose a flat image format for read-only partitions.

The graph also demonstrates significant performance gains on random writes, especially with small random writes, because LBS consolidates those writes and turn them into fewer and sequential ones.

Figure 5.11 demonstrates the performance of LBS against flat on trace-based tests. Each test was repeated three times and averaged. We set aside 160MB for each test to make sure that there are sufficient free blocks to prevent garbage collection from triggering. This experiment shows that we achieve huge performance gains when the workload has many small writes, as demonstrated by the Contact Database (about 17x). The email workload gains about 5x speed but does not have as impressive gains as others due to its high percentage of barrier operations, which forces a flush of the buffer and causes small writes for LBS.

5.5.2 Garbage collection

Garbage collection thread runs in the background and collects near empty clusters when the LBS data file is under pressure. The background process causes additional writes for collected clusters. These additional writes are inexpensive because they are done in the background during inactivity, and are sequential writes.

Table 5.5 shows the effect of *write amplification* due to LBS for the two workloads

trace	requested writes	LBS writes without GC	LBS writes with GC
Contacts	219124 blocks	219124 blocks 855 clusters	223593 blocks, 873 clusters (802 contiguous)
Email	216057 blocks	215972 blocks 843 clusters	220160 blocks 860 clusters (813 contiguous)

Table 5.5: Software-level write amplification due to garbage collection. 12% additional storage used for garbage collection.

that triggered garbage collection. The block size in this experiment is 1KB and the cluster size is 256KB. The first column is the number of writes requested by the guest operating system. The second column is the number of writes for LBS without GC. This is gathered with 100% over-provisioning so that the GC never triggers. The third column is the number of writes for LBS with GC. This is gathered with 12% over-provisioning and the GC does trigger. This table shows that when GC is not involved, LBS is able to consolidate the writes submitted by the operating system into fewer actual writes because applications sometimes overwrite stale data. When GC is involved, we have reasonable low write amplification and because those writes are mostly contiguous, the cost is relatively low.

As discussed in Section 5, a smart GC algorithm improves the efficiency of LBS especially when we have ample over-provisioning for GC. Using `sdperf`, we run experiments that issue random writes from the guest and ensures that ample GC activities are involved for the experiments. Figure 5.12 shows the effect of smart weighted GC when compared to naive GC. This graph demonstrates that when GC is involved, naive GC has much worse performance when GC over-provisioning is high, because the weighted GC generates more sequential writes; meanwhile when GC over-provisioning is low, the weighted GC has worse performance because it generates more GC writes than naive GC.

5.5.3 Encryption and integrity checking

Figure 5.11 also shows how encryption and integrity checking affect the performance of LBS. The level of security of LBS can be chosen between a fast but less secure Fletcher checksum or a slower but much more secure SHA-256 checksum. Because the level of indirection and the cost of issuing I/O commands are fixed regardless of block size, the larger the block size is, the more visible impact encryption has on the overall LBS performance. In our synthetic tests, we see an approximately 4-8% slowdown for using SHA-256 checksum on block size less than 4KB. This number is about 13-17% for block size of 256KB. For trace-based tests, we observe variable slowdowns for using encryption depending on how many barriers are present in the test. Email Client saw a 2% slowdown for having high barrier percentage meanwhile Android Boot experienced 35% slowdown.

5.6 Conclusion

In this chapter, we demonstrated why the characteristics of the SD card often mismatches with the workload of a virtual machine on mobile phones, and that it is unwise to store a flat virtual machine disk image directly on the SD card. We propose a solution using Logging Block Store to bridge such gap and ensure that many non-sequential writes from the guest are transformed into sequential writes on the SD card, thereby improving guest I/O performance. We also describe how reliability and security requirements of virtualied mobile devices can be met with LBS.

5.7 Acknowledgement for this chapter

This is a joint work with VMware while the author did an internship there. The author helped with creating benchmarks, analyzing the performance of LBS and optimizing the GC which prompted many changes to its architecture.

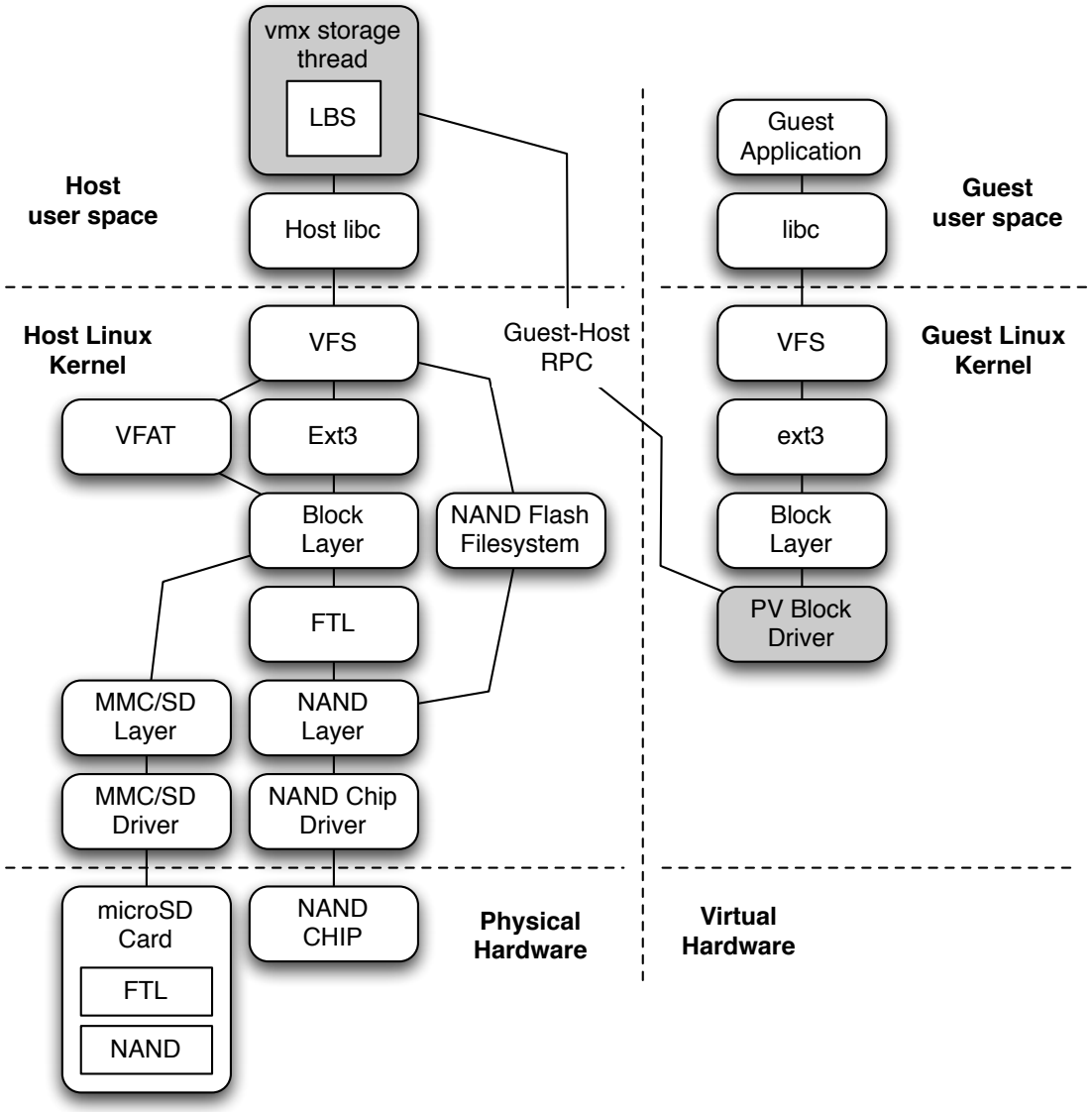

FIGURE 5.7: MVP storage architecture

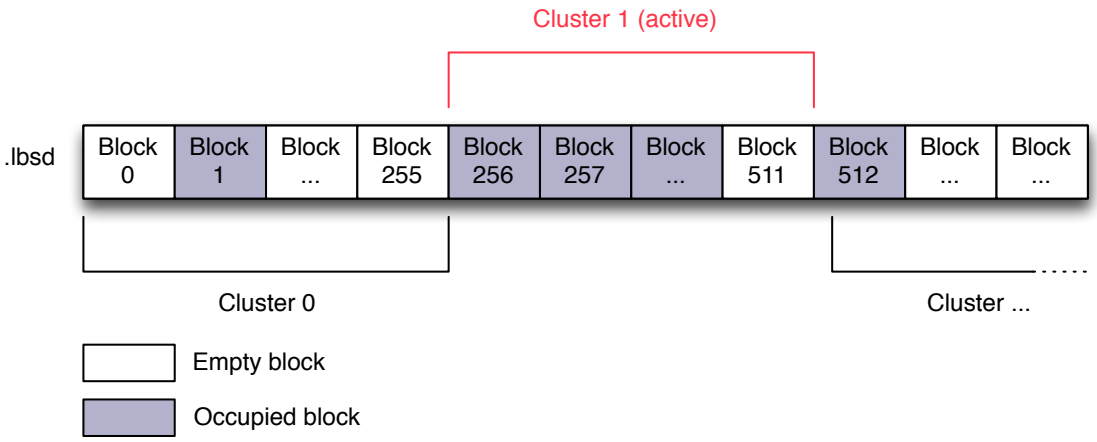

FIGURE 5.8: LBS data file format

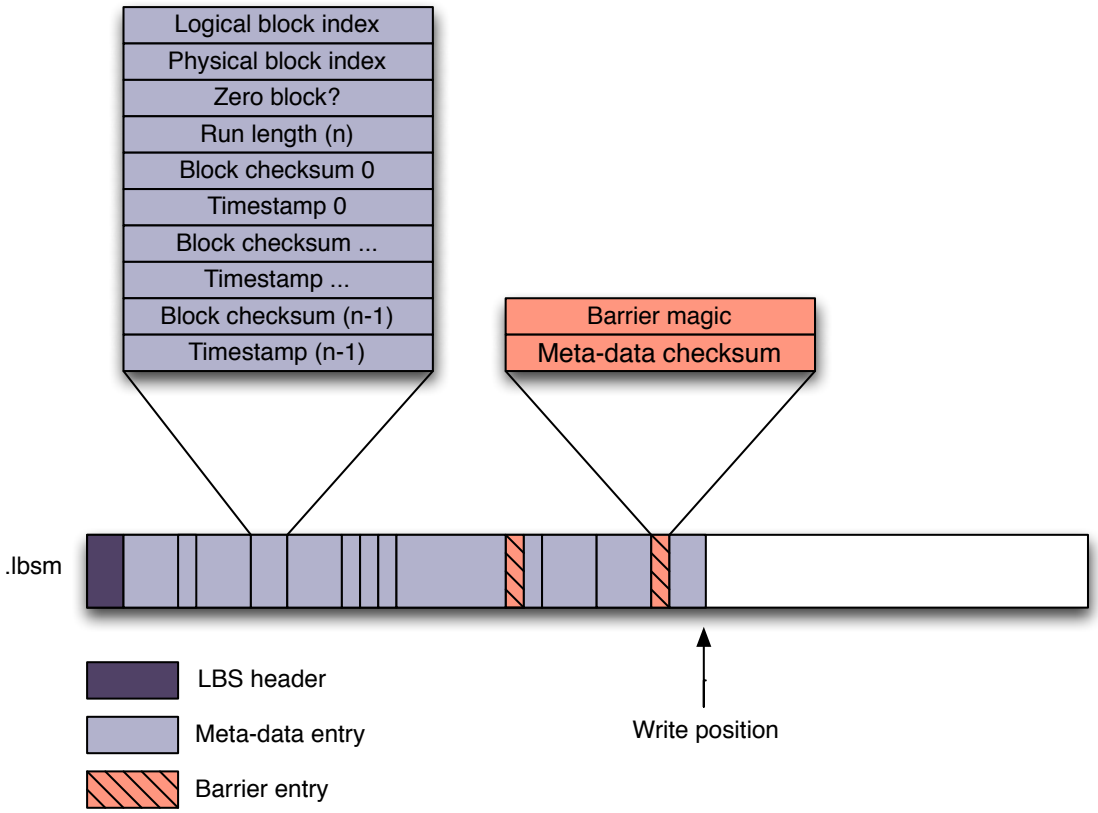

FIGURE 5.9: LBS meta file format

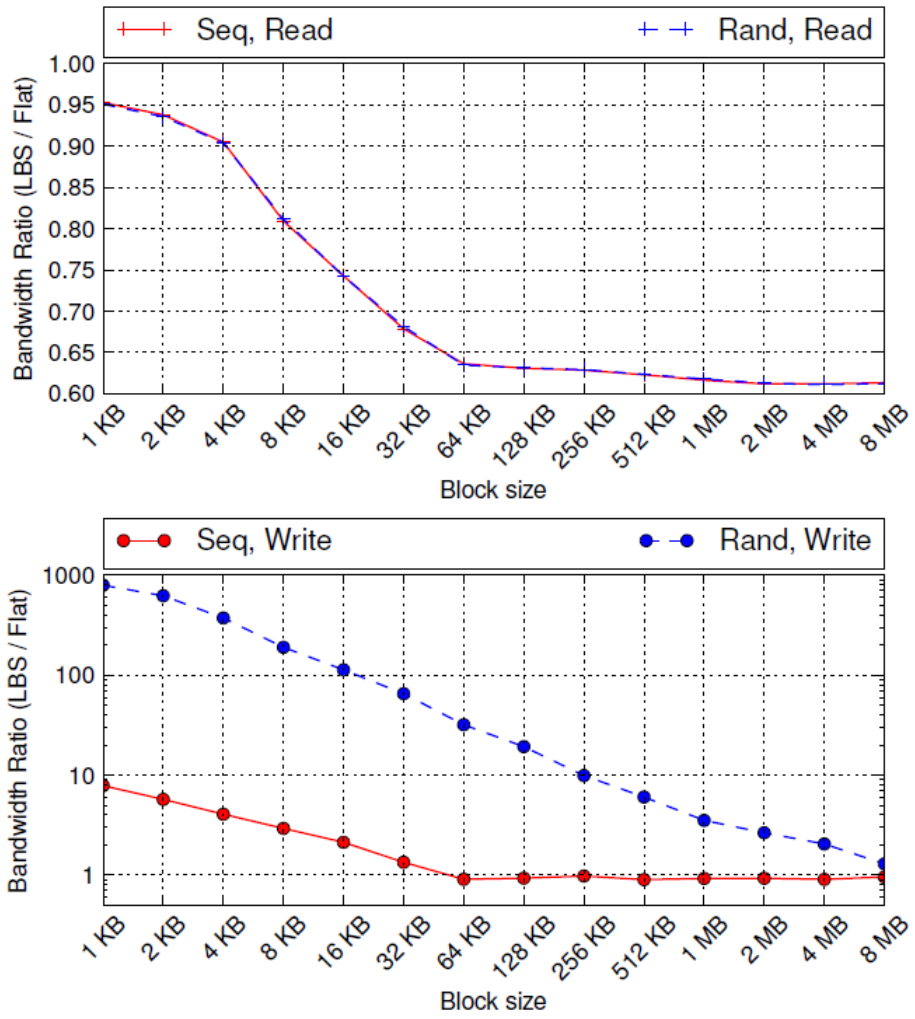

FIGURE 5.10: 8GB ADATA Class 6 SD card I/O bandwidth: Ratio between LBS and Flat file format

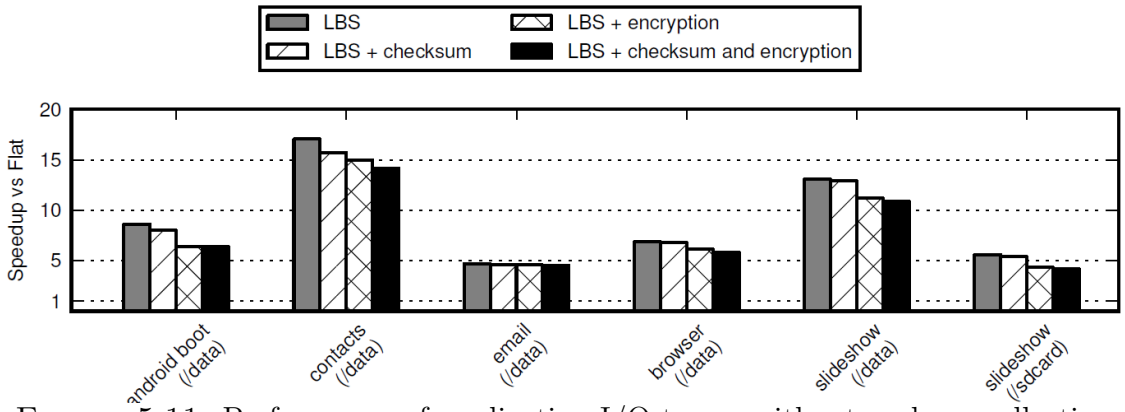

FIGURE 5.11: Performance of application I/O traces without garbage collection

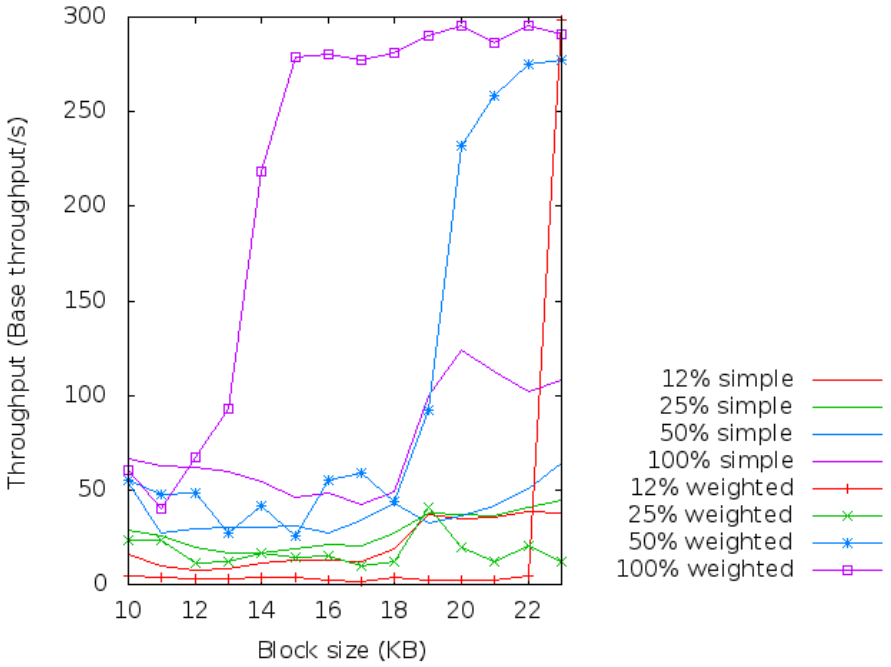

FIGURE 5.12: Performance comparison of naive GC against weighted GC using sdperf, with garbage collection, at different GC over-provisioning levels

6

Conclusions

6.1 Contributions

Virtualization on mobile and embedded systems is a new and interesting area of research. The different hardware and software ecosystem on these systems challenges us to discover techniques to circumvent many difficulties created by the inefficiencies of the I/O systems, or the relatively lack of hardware features to support virtualization functionalities such as CPU virtualization and record and replay. This thesis presents several contributions to existing virtual machine research.

We discussed several important findings regarding virtualization implementations:

1. It is possible to transparently virtualize without any guest modification on any architecture with hardware breakpoints. We use control flow analysis to detect sensitive instructions within the virtual machine, and trap them using hardware breakpoints. We built a prototype proof of concept on both x86 and ARM architecture to demonstrate its portability and feasibility.

2. Using guest introspection, which is the ability to inspect on virtual machines to

gain guest specific information such as processes, signals and memory sharing without having to modify the guest, we are able to efficiently implement record and replay without using hardware branch counters.

3. SD cards and similar low-cost flash media favors sequential writes and has terrible performance on non-sequential writes. This characteristic must be taken into account when designing virtualized storage subsystems, such as a virtual machine disk image, on those devices.

6.2 Future work

In the long run, ARMv8 and its Virtualization Extension (VE) will be ubiquitous, however our work may prove to be useful on some other classes of devices. Internet of Things (IoT) devices, for example, may feature smaller and simpler processors and may be an interesting area to consider core virtualization in.

It is possible to take our record and replay implementation and implement useful virtualization features built upon it. For example, we could construct a fault tolerant mobile operating system, or a debugger for mobile OS. We could also use the record and replay functionality to impose taint tracking and security properties on mobile operating systems by running a shadowed copy of the running OS and selectively replaying the trace with different inputs.

For our LBS work, it is worth exploring how to secure the LBS encryption key from host compromises using hardware security measures such as Trustzone. Also, although we only discussed storage virtualization as an example, virtualizing other peripheral devices on mobile phones is not trivial. Network device virtualization for example, may involve telephony and carrier issues. GPS device virtualization, on the other hand, might contain security and privacy problems.

It would also be interesting to try to apply the techniques we proposed to a

variety of other emerging platforms and embedded systems. MIPS for example, is a valid target for our implementation due to having hardware breakpoints and no hardware branch counters.

In addition to the technical challenges we explored and discussed in this thesis, our work may prove useful to hardware designers of future CPUs on how to efficiently support possible virtualization use cases with minimal efforts. For example, CPU virtualization can be trivialized by eliminating sensitive instructions and have all of those instructions fault instead. Also, cheap context and mode switches would significantly improve VMM performance and reduce the amount of optimization required to achieve optimal performance.

Another notable trend of everyday computing is the prevalence of flash storage devices. Traditional filesystems and storage structures are often inadequate at handling flash storage devices' hardware characteristics. Log structured filesystems or journaling filesystems are often required to provide efficient storage capabilities to system or application software. System level changes, such as having FTL aware filesystems or operating systems, may potentially become a hot research area.

In conclusion, we are able to provide efficient virtualization functionalities despite hardware hurdles and making virtualization one step closer to a solution for isolation, encapsulation and security needs for everyday computing.

Bibliography

(2001), "ARM Architecture Reference Manual, C9.3," *ARM Information Center*.

Adams, K. and Agesen, O. (2006), "A comparison of software and hardware techniques for x86 virtualization," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 2–13, ACM.

Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. (2008), "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference*.

Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. (2011), "Cells: a virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 173–187, New York, NY, USA, ACM.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003), "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, 37, 164–177.

Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., and Zoppis, B. (2010), "The VMware mobile virtualization platform: is that a hypervisor in your pocket?" *SIGOPS Oper. Syst. Rev.*, 44, 124–135.

Bartels, G. and Mann, T. (2001), "Cloudburst: A Compressing, Log-Structured Virtual Disk for Flash Memory," Tech. Rep. 2001-001, Compaq Systems Research Center.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazieres, D., and Kozyrakis, C. (2012), "Dune: safe user-level access to privileged cpu features," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 335–348, USENIX Association.

Bellard, F. (2005), "QEMU, a Fast and Portable Dynamic Translator." in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46.

Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. (2010), "The Turtles Project:

Design and Implementation of Nested Virtualization." in *OSDI*, vol. 10, pp. 423–436.

Bergmann, A. (2011), "Optimizing Linux with cheap flash drives," *Linux Weekly News*.

Birrell, A., Isard, M., Thacker, C., and Wobber, T. (2007), "A design for high-performance flash disks," *SIGOPS Operating Systems Review*, 41, 88–93.

Bouganim, L., Jónsson, B., and Bonnet, P. (2009), "uFLIP: Understanding Flash IO Patterns," in *Conference on Innovative Data Systems Research*.

Bressoud, T. C. and Schneider, F. B. (1996), "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, 14, 80–107.

Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. (2012), "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Transactions on Computer Systems (TOCS)*, 30, 12.

Chow, J., Lucchetti, D., Garfinkel, T., Lefebvre, G., Gardner, R., Mason, J., Small, S., and Chen, P. M. (2010), "Multi-stage replay with crosscut," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pp. 13–24, New York, NY, USA, ACM.

Desnoyers, P. (2013), "What systems researchers need to know about NAND flash," in *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*, pp. 6–6, USENIX Association.

Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. (2002), "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, 36, 211–224.

Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. (2008), "Execution replay of multiprocessor virtual machines," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 121–130, ACM.

El Maghraoui, K., Kandiraju, G., Jann, J., and Pattnaik, P. (2010), "Modeling and simulating flash based solid-state disks for operating systems," in *WOSP/SIPEW International Conference on Performance Engineering*.

Fletcher, J. G. (1982), "An Arithmetic Checksum for Serial Transmissions," *IEEE Transactions on Communications*, 30, 247 – 252.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. (1996), "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 1996 USENIX Security Symposium*.

Goldberg, R. P. (1974), "Survey of virtual machine research," *Computer*, 7, 34–45.

Hwang, J.-Y., bum Suh, S., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R. (2008), "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pp. 257–261.

Jeong, S., Lee, K., Lee, S., Son, S., and Won, Y. (2013), "I/O stack optimization for smartphones," in *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pp. 309–320, USENIX Association.

Jiang, S., Chen, F., and Zhang, X. (2005), "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." in *USENIX Annual Technical Conference, General Track*, pp. 323–336.

Kim, H., Agrawal, N., and Ungureanu, C. (2012), "Revisiting storage for smartphones," *Trans. Storage*, 8, 14:1–14:25.

King, S. T., Dunlap, G. W., and Chen, P. M. (2003), "Operating System Support for Virtual Machines." in *USENIX Annual Technical Conference, General Track*, pp. 71–84.

King, S. T., Dunlap, G. W., and Chen, P. M. (2005), "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 1–1.

Lawton, K. P. (1996), "Bochs: A Portable PC Emulator for Unix/X," *Linux J.*, 1996.

Lee, D., Wester, B., Veeraraghavan, K., Narayanasamy, S., Chen, P. M., and Flinn, J. (2010), "Respec: efficient online multiprocessor replayvia speculation and external determinism," *SIGPLAN Not.*, 45, 77–90.

Linaro (2001), "Flash Card Survey," .

Liu, H., Jin, H., Liao, X., Hu, L., and Yu, C. (2009), "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pp. 101–110, New York, NY, USA, ACM.

Nath, S. and Gibbons, P. B. (2010), "Online maintenance of very large random samples on flash storage," *The VLDB Journal*, 19, 67–90.

Rajimwale, A., Prabhakaran, V., and Davis, J. D. (2009), "Block management in solid-state devices," in *USENIX Annual Technical Conference*.

Rosenblum, M. and Ousterhout, J. K. (1991), "The design and implementation of a log-structured file system," in *ACM Symposium on Operating Systems Principles*.

Saxena, M. and Swift, M. M. (2010), "FlashVM: virtual memory management on flash," in *USENIX Annual Technical Conference*.

Biography

About the author:

Name: Bi Wu

DOB: Feb 9 1985, Shenyang, China.

Degrees earned:

1. Ph.D. Duke University
2. M.S. Duke University
3. B. Engineering. National University of Singapore

The author is going to work at VMware after graduation.